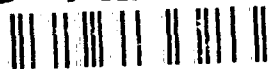


AD-A235 779



Educational Materials
CMU/SEI-89-EM-1

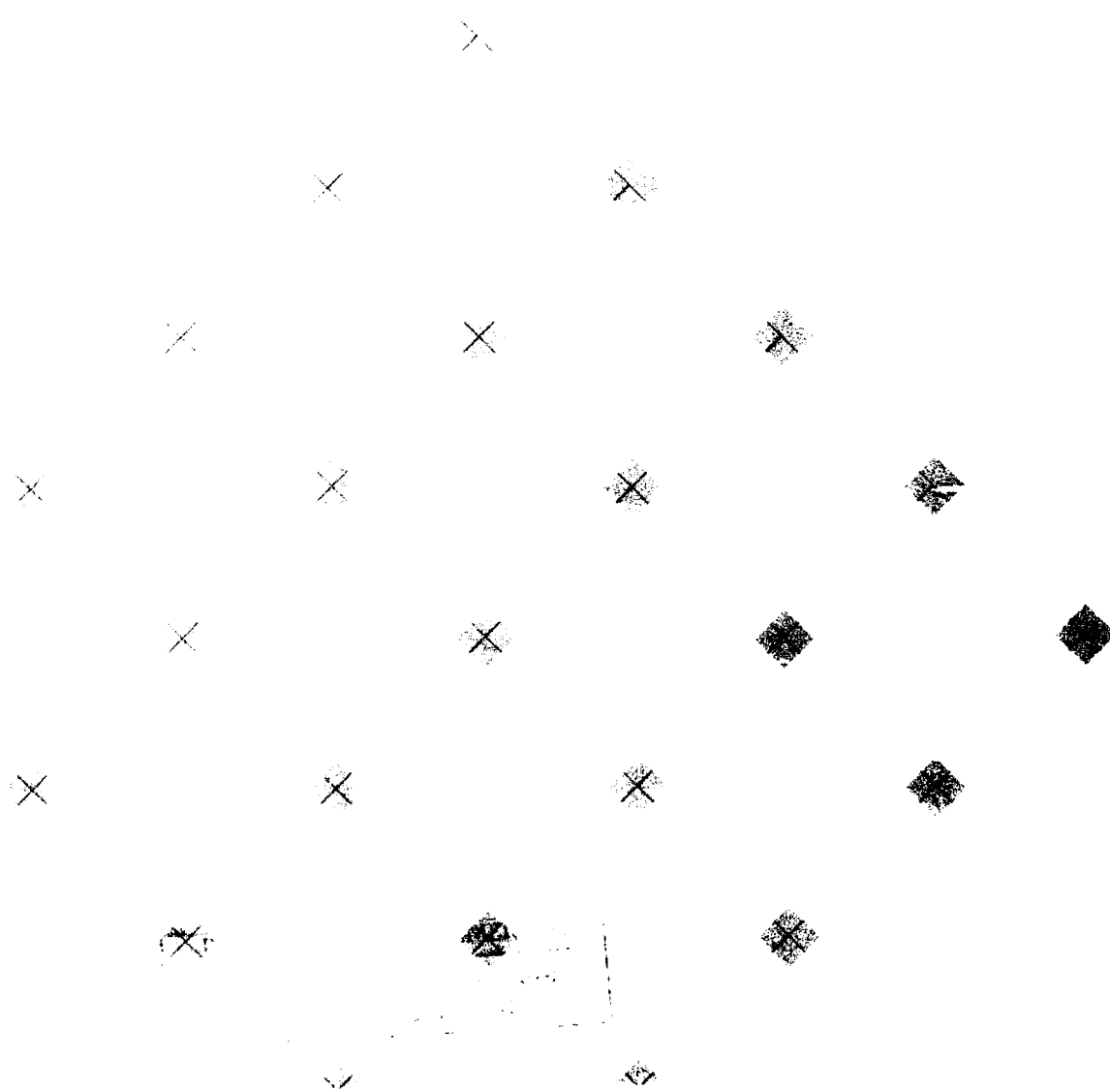
2

TIC



Software Maintenance Exercises for a Software Engineering Project Course

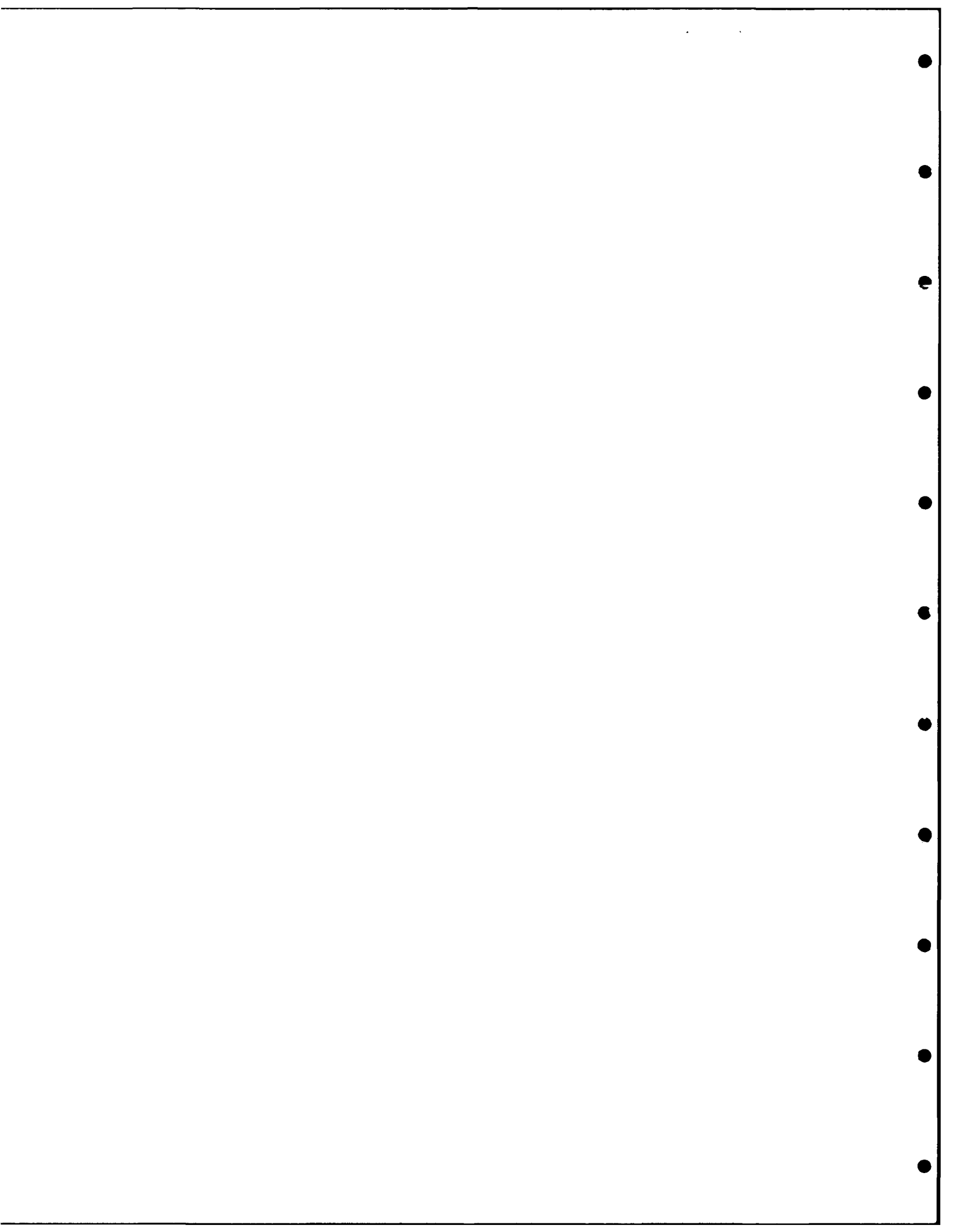
Charles B. Engle, Gary Ford, Tim Korson
February, 1989



91-00320



91 5 22 054



Educational Materials

CMU/SEI-89-EM-1

February 1989

Software Maintenance Exercises for a Software Engineering Project Course



Charles B. Engle

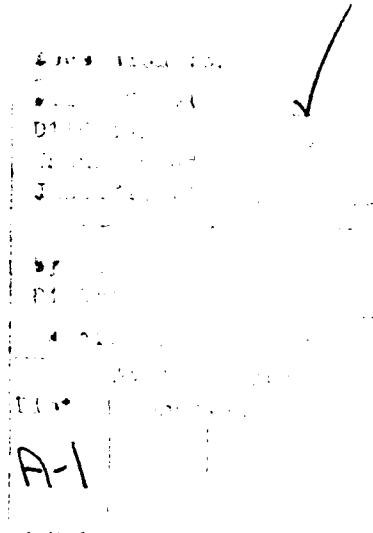
U.S. Army SEI Resident Affiliate

Gary Ford

SEI Undergraduate Software Engineering Education Project

Tim Korson

Clemson University



Approved for public release.
Distribution unlimited.

Software Engineering Institute

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER


Karl H. Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1989 by Carnegie Mellon University.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1. Introduction	1
2. Software Maintenance	2
3. The DASC Software System	4
4. Software Maintenance Exercises	6
4.1. Develop Documentation Standards	6
4.2. Develop Configuration Management Plan	7
4.3. Install and Test the DASC Software	8
4.4. Update Documentation after Porting	9
4.5. Develop Regression Test Plans	10
4.6. Discrepancy Report 1: Unexpected Constraint Exception	10
4.7. Discrepancy Report 2: Apparent Parameter Mode Error	12
4.8. Discrepancy Report 3: File Name Length Errors	13
4.9. Discrepancy Report 4: Empty Input File Error	14
4.10. Discrepancy Report 5: Unreachable Code	14
4.11. Code Reviews	15
4.12. Change Request 1: Improved Flaw and Style Messages	16
4.13. Change Request 2: User Interface, Version 1	16
4.14. Change Request 3: User Interface, Version 2	17
4.15. Change Request 4: User Interface, Version 3	17
4.16. Change Request 5: Add Page Headers to Reports	18
4.17. Change Request 6: Add Line Numbers to Flaw Reports	19
4.18. Change Request 7: Allow User-Specified Style Parameters	19
Annotated Bibliography	21
Appendix 1. Project Team Roles	24
Appendix 2. Distribution Diskette Contents	25
 Attachment 1. Discrepancy Reports and Change Requests	
Attachment 2. DASC Documentation	
Attachment 3. Diskette Order Form	

Software Maintenance Exercises for a Software Engineering Project Course

Abstract

Software maintenance is an important task in the software industry and thus an important part of the education of a software engineer. It has been neglected in education, partly because of the difficulty of preparing a software system upon which maintenance can be performed. This report provides an operational software system of 10,000 lines of Ada and several exercises based on that system. Concepts such as configuration management, regression testing, code reviews, and stepwise abstraction can be taught with these exercises.

1. Introduction

Because many if not most computer science majors go on to careers involving software development, a project-oriented course in software engineering can be very valuable in the curriculum. One of the goals of the Undergraduate Software Engineering Education Project at the Software Engineering Institute (SEI) is to provide instructors and students with guidelines and materials for such a course.

Toward that end, in 1987 the SEI published the technical report *Teaching a Project-Intensive Introduction to Software Engineering* [Tomayko87b]. This report identified four different models for such a course and then presented detailed guidelines for one of them, the "large project team" model. This model requires 10 to 20 students organized as a software project team, with different students playing different roles (such as principal architect, project administrator, configuration manager, quality assurance manager, test and evaluation engineer, documentation specialist, and maintenance engineer). The instructor plays the role of project manager. The student roles are defined in Appendix 1.

With such a course structure, not every student writes code; in fact, very few of the students write code. Instead, the students experience directly or indirectly all the aspects of a software development project, and that is what makes such a course a software engineering course rather than simply an advanced programming or group programming course.

A long-standing difficulty with such a course is that there is almost never enough time to develop a piece of software from scratch and then have the students do some maintenance on it. Many instructors are unaware of the importance and methods of software maintenance, and they often do not even include the subject in their course syllabus. Even instructors who do want to teach maintenance often cannot devote the time to finding or developing a system for the students to maintain. Since software maintenance is a fact of life in the software indus-

try, it is important for students to have experienced it and learned some of the known techniques.

The intent of this report is to make teaching software maintenance more feasible in a software engineering project course. This report provides a operational software system, called the *Documented Ada Style Checker (DASC)*, (described in Section 3); a reasonable set of documentation for the system; and specific exercises with guidelines for the instructor. Altogether, the materials could be the basis for a semester-long course. Individual exercises might be assigned as part of other courses, including a project course based primarily on new development.

The software system is written in Ada. For instructors and students new to Ada, there is a great advantage in designing a course around maintenance rather than new development. Students are able to work on a much larger system, and thus experience many more Ada constructs, than would be the case if they had to learn the language in parallel with developing code. In general, analysis is easier than synthesis in engineering.

2. Software Maintenance

The term *software maintenance* is generally used to mean changing a program in order to correct errors, improve performance, adapt to a changing environment, or provide new capabilities. Some consider this to be an abuse of the term *maintenance* and suggest other terminology, including *software evolution* and *post-deployment software support*. However, the term *maintenance* is widely used and understood, so we will use it here also.

In simple models of software development, such as the common *waterfall model*, maintenance is considered to be an activity separate and different from development. From a software engineering standpoint, however, it is better to view maintenance as involving the same activities as those of development (such as requirements analysis and specification, design, implementation, and testing) but performed with different constraints. The most significant of those constraints is the existence of a body of code and documentation that must be incorporated into the new version of the system.

Usually the cost of modifying the existing system is less than that of creating an entirely new system with the desired new functionality. This is the fundamental justification for software maintenance. However, it is the responsibility of the software project manager to recognize when this is not the case and that the existing system should be retired and a new system produced.

Swanson defines three categories of software maintenance [Swanson76]:

- Perfective: modifications requested by the user (usually because of changing or new requirements) or by the programmer (usually because of the discovery of a better way to implement part of the system).
- Adaptive: modifications necessitated by changes in the environment in which the program operates (including transporting the program to a different computer system).
- Corrective: modifications to correct previously undiscovered errors in the program.

The exercises in this report include some in each of these categories.

There are relatively few techniques or methods specifically for software maintenance as compared to new software development. There are, however, a few software engineering techniques whose usefulness can be demonstrated especially well through maintenance efforts. Four that we recommend to instructors and students are:

- Software Configuration Management
- Regression Testing
- Code Reviews
- Stepwise Abstraction

Software configuration management encompasses the disciplines and techniques of initiating, evaluating, and controlling change to software products during and after the development process. The students should be required to develop and adhere to a software configuration management plan as part of the course. The software system described in this report consists of approximately 10,000 lines of code (in 63 separately compilable program units) and nine documents. When the students are working on the changes to both program and documentation, especially when different students are working on different changes, careful configuration management is essential to the project. Therefore one of the first recommended exercises is the development of the configuration management plan. Additional information on configuration management may be found in [Tomayko86] and [Tomayko87a].

Regression testing is defined as "selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets its specified requirements" [IEEE83]. Some of the exercises require major changes to the software system and therefore call for substantial retesting, perhaps involving the entire test suite. Other exercises require rather minor changes, and a single, simple retest may be sufficient. One of the first recommended exercises is the development of regression test plans. Additional information on regression testing may be found in [Collofello88b].

Code reviews offer an opportunity for software developers to discover errors or inefficiencies in their code earlier in the development process. Their use is an application of a fundamental principle of engineering: it is almost always less costly to find and correct an error early in the process rather than late. They are becoming increasingly common in industry, so students should learn at least one form of review in a software engineering course. Reviews can be conducted in a number of different ways; a good introduction for the instructor may be found in [Collofello88a].

Stepwise abstraction is a technique pioneered by IBM Federal Systems Division (now Systems Integration Division). It is used to recover the high level design of a system in the absence of design documentation. The design can then be used to plan program changes. Britcher and Craig describe the process as follows [Britcher86]:

"From the source code, the designer abstracted the module design and recorded it using PDL [Process Design Language]. Choosing the level of abstraction based on the module, the designer determined the change required. Often this was an iterative process; the designer abstracted a detailed design from the code, then generated

another less detailed (yet still precise) abstraction from that design. The iteration continued until the designer was comfortable with the level of abstraction."

Some of the exercises in this report can take advantage of this technique (for example, exercises 4.16, 4.17, and 4.18). For the DASC system, which is reasonably well structured and makes good use of Ada packages, this abstraction is quite straightforward. However, because it is a powerful and useful technique, we strongly recommend using it. The instructor may want to lead a classroom discussion to introduce the process.

3. The DASC Software System

The *Documented Ada Style Checker (DASC)* software system examines syntactically correct Ada programs and reports on their adherence to predefined style conventions. Examples of the style conventions examined are:

- Case of characters in reserved words and object identifiers
- Consistency of indentation to show program control structures
- Use of blank lines to set off program blocks
- Subprograms too short or too long
- Control structures or packages nested too deeply
- Use or lack of use of Ada-specific features

The style checker produces two kinds of reports, called a *flaw* report and a *style* report. The former identifies specific statements in the program that violate style conventions, and the latter is a quantitative summary of the program's style.

The system was originally developed on a Data General computer system and later ported to a DEC VAX VMS system. It was then placed in the Ada Software Repository, and hence in the public domain. (For information on the repository, see [Conn87].)

In the spring of 1988, Prof. Linda Rising of Indiana University-Purdue University at Fort Wayne (IPFW) selected the system as the basis of a software engineering project course. Her students were given the task of porting the system to the university's VAX and providing a reasonable set of documentation for it (hence the name *documented Ada style checker*).

The student documentation consists of the following documents:

1. Requirements Document
2. Preliminary Design
3. Detailed Design
4. Documentation Standards and Guidelines
5. Coding Standards
6. Quality Assurance Plan
7. Test Plan
8. Configuration Management Plan
9. User Manual

The requirements and design documents (items 1-3), having been produced from the source code, clearly are not as complete as one would expect in a real software development project. However, they do provide a starting point for maintenance exercises, including maintenance of the documents themselves.

The documentation and coding standards and the three plan documents (items 4-8) were used by Prof. Rising and her students to guide their project. These documents reflect both development and maintenance activities. Some of the first exercises in this report involve updating these documents to reflect new maintenance activities.

The user manual (item 9) describes the use of the system in its IPFW implementation. Much of this document is specific to VAX VMS systems and some specific to IPFW. Porting the software to another computer system will require extensive revision of this document. In particular, porting the system will require redesign and reimplementations of the user interface, the description of which constitutes a major part of this document. Some of the exercises in this report are based on the development of a new user interface (see exercises 4.13, 4.14, and 4.15).

In the summer of 1988, Prof. Rising provided the software and student documentation to the SEI for further development and release as teaching support materials. Prof. Tim Korson of Clemson University, who was a visiting scientist at the SEI, succeeded in porting the system to a VAX VMS system and to a Zenith Z248/MS-DOS system running Alsys Ada. He also developed the first maintenance exercises. Subsequently, Maj. Chuck Engle, a U.S. Army resident affiliate at the SEI, ported the system to a VAX Ultrix system running Verdex Ada, and he developed some additional exercises. SEI staff developed other exercises, and edited and formatted the student documentation.

It is interesting to note that each of the three Ada compilers on the three different computer systems reported a different set of errors and warnings when the program was compiled. Although they were not reported by all compilers, each error has been documented as a discrepancy report and the correction of the error appears in this report as an exercise (see exercises 4.6 through 4.10).

The SEI has prepared distribution diskettes containing the DASC system source code, the student documentation, tools, and the test suite. These may be ordered in two formats, Macintosh 3.5" 800K byte diskettes and IBM PC 5.25" 1.2M byte diskettes. The documents are available in three formats: as *Microsoft Word* documents and *MacWrite* documents for the Macintosh, and as text-only documents for any system. A description of the contents of the distribution diskettes appears in Appendix 2, and a diskette order form appears at the end of this report.

We assume that many users of this software will want to upload the system from a PC to a VAX VMS computer system. To help in this process, the distribution diskettes also include two command files that can be used to translate between the relatively long file names of the VMS version and the eight character names required under MS-DOS. In addition, the diskettes include a file giving the required compilation order of all the program units.

We do not present this artifact as a model of good coding style, design, or documentation. In fact, if the style checker is run on itself, it reports many problems. There are some fairly obvious design improvements that could be made. The documentation is reasonable although not complete, and no formal analysis, design, or documentation technique was used.

In summary, one might say that this artifact seems to be a fairly representative example of existing software systems. This is not necessarily bad, because a valid educational objective might be to expose students to "the real world."

4. Software Maintenance Exercises

The exercises described in this section are presented in roughly the order in which they might be assigned to the students. Most of the exercises, however, are relatively independent, and instructors should feel free to select those that are most appropriate for their particular courses.

We recommend that the first five exercises, which deal with project management, be included in all courses. In most cases, these exercises can be assigned to different students or groups of students, those playing the roles of documentation specialist, configuration manager, test and evaluation engineer, and verification and validation engineer (see Appendix 1 for descriptions of these roles).

Exercises 4.6 through 4.10 are presented as *discrepancy reports* and exercises 4.12 through 4.18 as *change requests*. These reports and requests, formatted as one page forms, appear in Attachment 1 and also on the distribution diskettes. We recommend that instructors tailor these to their circumstances, print copies, and submit them to the student Change Control Board for action. The project leader and the configuration manager can assign responsibility for making the appropriate changes in the code and the documentation.

Exercise 4.11 can be used to introduce the concept of a *code review*. The exercise identifies a module in the code for which a careful inspection should uncover opportunities for improving the code. The result will be an additional change request. Once the students are familiar with code review, they should be asked to conduct them for their own code.

4.1. Develop Documentation Standards

Exercise

Select word processing or document processing software to be used to develop and maintain project documentation. Transfer the documents from the distribution diskette to the appropriate computer system. Modify the document *Documentation Standards and Guidelines* to reflect local requirements and capabilities.

Information for Instructors

The objective of this exercise is to introduce the students to the system documentation and to the idea of maintaining documentation along with the code. Too often in the academic environment, documentation is an afterthought.

It is likely that the instructor will select the appropriate documentation software. If possible, allow the students to use the same computer system for code development and documentation. This can give more of the feel of an integrated programming environment in which it is easy to manipulate all kinds of software work products in a coordinated way.

In any case, the documentation specialist role should be assigned to a student with good knowledge of the word processing software being used. The instructor should ensure that the documentation standards developed by the student are reasonable. That is, adherence to the standards should be easily within the capabilities of all students. An in-class formal review of the document can be used to help identify problems with the standards.

The documentation standards supplied with the DASC system reflect the fact that IPFW students used Macintosh systems with *Ready, Set, Go!* and *Excelerator* software. The documents themselves were prepared for distribution on a Macintosh with *Microsoft Word* and *MacWrite* software. If such a system is available, very little change in the documentation standards should be required.

Otherwise, we recommend that the text-only versions of the documents be the basis for project documents. Many more changes in the documentation standards will then be required.

Only one or two students are likely to be involved in this exercise. Therefore it can be done in parallel with other exercises.

4.2. Develop Configuration Management Plan

Exercise

Using the existing *DASC Configuration Management Plan* as a basis, develop an appropriate configuration management plan.

Information for Instructors

The objective of this exercise is to introduce the students to the basic concepts of configuration management, including the *configuration management plan* document and the *change control board*. It is important that the document be approved and the change control board be in place before the code maintenance exercises be attempted. This will not only help make the code installation (see the next exercise) more well-defined, but it also helps instill in the students the idea of *plan first, execute later*.

The existing plan will need only minor modifications when used in a course in which different students play different project roles. Some modifications that will certainly be needed are the following:

- Change the date mentioned in section 2.3.4.

- Change the names of the project directories as needed for your particular computer system. References to these directories appear in sections 2.3.2, 3.1.3, 3.1.4, 3.2.5, 3.3, and appendices 1 and 2.
- Provide an appropriate list of test and support tools that will be used by the students. This list is referenced in section 2.3.3.
- Choose an appropriate file name convention, as described in section 3.1.5.2.
- Choose an appropriate file protection convention, as described in section 3.2.6.

4.3. Install and Test the DASC Software

Exercise

Transfer the source code from the distribution diskette to an appropriate computer system. Structure the code directory or directories as specified in the configuration management plan. Compile each program unit, using the compilation order defined on the distribution diskette. Record all compiler error and diagnostic messages.

Run the program on the entire test suite. Record any discrepancies for consideration by the Change Control Board.

Information for Instructors

The objectives of this exercise are to make the software system operational in preparation for the later exercises and to give the students the experience of trying to install a system that they have not themselves written. The absence of a detailed installation guide will require the students to improvise. The instructor may wish to have the students write such a guide for their particular computer system after the installation.

The DASC source code on the distribution diskette is known to have some errors (see exercises 4.6 through 4.10). Depending on the Ada compiler used, students may discover some of these known errors (but probably not all of them), and they may also discover some additional errors.

In many cases, the students may see error messages related to wrong compilation order, typographical errors in commands, or just misunderstanding of the Ada system they are using. These errors should be corrected immediately so that the installation can continue.

Error and diagnostic messages from the compiler should be recorded as discrepancy reports. The reports should be submitted to the student Change Control Board for action rather than being corrected immediately. Errors uncovered by running the test suite should also be recorded as discrepancy reports.

4.4. Update Documentation after Porting

Exercise

Revise all documentation as required to reflect the system after installation on the new computer system and to bring all documents into compliance with the documentation standards.

Information for Instructors

The objective of this exercise is to continue to instill in the students the idea that the documentation is an integral part of the system, and that any maintenance effort must include documentation maintenance.

Instructors should take care to ensure that documentation changes are handled like code changes, meaning that they are considered by the Change Control Board and the configuration manager. Note that the discrepancy report form and change request form have places for recording the documentation affected by changes.

Some documentation revisions were detailed in exercise 4.2. Other places where revisions will be necessary are:

- Section 5.1 of the Test Plan mentions the names of the test files and the directory in which they reside. These should be changed as required by the computer system being used by the students.
- Modify the user manual to reflect the user interface, assuming that the students are not using the VAX VMS interface supplied on the distribution diskette.

This exercise can also be used to correct some known problems with the original documentation distributed with this report. These problems include:

- Section 3 of the Test Plan shows the relationships between a requirement and the test case for that requirement. The table mentions only four test cases, when in fact there are seven test cases supplied on the distribution diskette.
- The Documentation Standards and Guidelines describe a standard for representation of acronyms in documents. This standard is not followed in the documents on the distribution diskette. Either the documents or the standard should be changed.
- The Preliminary Design Document and the Detailed Design Document do not contain revision history sections. These should be added.

The documentation specialist will have primary responsibility for this exercise. Other students may begin some of the later exercises in parallel with the documentation revisions.

4.5. Develop Regression Test Plans

Exercise

Section 5.2 of the Test Plan document describes how regression testing is to be performed. Modify this section as necessary.

Information for Instructors

The objective of this exercise is to introduce the concept of regression testing. It is likely that the students will not have encountered this concept in earlier programming courses. Therefore it is important for the instructor to spend some time discussing the reasons for doing regression testing and the importance of maintaining the test plan so that regression testing may be done properly when needed.

The student designated to be the project test engineer should have continuing responsibility to keep section 5.2 of the test plan up to date. As new system requirements are approved by the Change Control Board, the test engineer should also revise section 3 of the test plan. The instructor, as project manager, should be responsible for ensuring that this student keeps the test plan current. An in-class review of the parts of the test plan related to regression testing can be helpful.

4.6. Discrepancy Report 1: Unexpected Constraint Exception

Exercise

During execution, an exception is raised in the procedure `ENTERING_BLOCK_STRUCTURE`. Identify and correct the error that causes this exception to be raised.

Information for Instructors

The objective of this exercise is to give the students experience correcting an error that raises an exception in Ada. Instructors may want to point out that the error causing this exception might go undetected in most earlier programming languages.

This is an actual bug that was discovered when porting the system to the Alsys compiler on a PC/AT-class machine. The problem did not occur in the VAX VMS/DEC Ada environment. If the Alsys compiler is available, the strategy actually used to identify the error, described below, may be useful for students. In other cases, it will be necessary for the instructor to identify the exception and the statement that causes it to be raised.

Since an exception is raised in a `when others` statement, the first step in the problem is to determine which exception is being raised. This is easily accomplished by adding `when state-`

ments for all possible exceptions. Alternatively, some compilers provide a way to determine the current exception name. After doing this, the exception will be identified as a constraint error.

The students are likely to make some common errors when following this strategy, especially if they are new to Ada. The instructor might expect the following errors:

- If students test for `when IO_EXCEPTION.anything`, they should be sure to add a `with IO_EXCEPTIONS;` statement when compiling the package.
- If students block off a portion of the program and add:

```
EXCEPTION
  when constraint_error => Put_line(" Error on line xxx ");
consider propagating the exception with:

Raise
```

To identify which statement is causing the constraint error, the procedure can be divided into blocks, each of which has its own exception handler. The block containing the error can then be subdivided into smaller blocks until the statement causing the error is identified. That statement will be found to be:

```
CURRENT_STATUS.PROCEDURE_NEST_LEVEL :=
  CURRENT_STATUS.PROCEDURE_NEST_LEVEL +1
```

This approach to the exercise requires a knowledge of the classical *divide-and-conquer* debugging strategy, and how to use the Alsys compiler, but does not require an in-depth knowledge of Ada. A brief introduction to the syntax and semantics of exception handling in Ada along with access to a reference manual should be sufficient for an experienced programmer.

A constraint error indicates that the variable is being given a value that is not valid for its type. Because the new value is the former value plus one, the former value must be incorrect. An examination of all references to this variable will show that it and two other related variables are never initialized. Each of these variables (`PACKAGE_NEST_LEVEL`, `CONTROL_NEST_LEVEL`, and `PROCEDURE_NEST_LEVEL`) is used as a nesting level counter, and each should start at 0.

Once the students have determined how these variables are used, they must determine how and where to initialize them. All three are fields in the variable `CURRENT_STATUS`, which is declared in procedure `STYLE_CHECKER`. An examination of that procedure shows two ways to accomplish the initialization.

The first way is to write assignment statements in the body of the procedure:

```
CURRENT_STATUS.PACKAGE_NEST_LEVEL := 0;
CURRENT_STATUS.CONTROL_NEST_LEVEL := 0;
CURRENT_STATUS.PROCEDURE_NEST_LEVEL := 0;
```

The second way is to give default initial values to these fields in the declaration of the record type itself:


```

type STATUS_RECORD is record
    ...
    PACKAGE_NEST_LEVEL : TOKENIZER.LINE_INDEX_RANGE := 0;
    CONTROL_NEST_LEVEL : TOKENIZER.LINE_INDEX_RANGE:= 0;
    PROCEDURE_NEST_LEVEL : TOKENIZER.LINE_INDEX_RANGE:= 0;
    ...
end record;

```

4.7. Discrepancy Report 2: Apparent Parameter Mode Error

Exercise

The compiler reports that *out* mode parameters in two procedures are not given values. Determine whether this error is a result of the parameters' modes being incorrectly specified, whether those parameters are not needed at all, or whether the parameters should have been given values in the bodies of the procedures. In this latter case, supply the code to give the parameters appropriate values.

The errors are reported for procedures `CREATE_DICTIONARY` and `TOKEN_IS_FOUND`, both of which are defined in package `DICTIONARY_MANAGER`.

Information for Instructors

The objective of this exercise is to give the students experience in a situation where the original developers of the code wrote some obviously unusual code but did not document their reasons for doing so. In this case, the students should deduce that the developers were leaving a "hook" for additional functionality that was never added.

The code for the package in question is:

```

package body DICTIONARY_MANAGER is

    procedure CREATE_DICTIONARY(DICTIONARY_KIND : in DICTIONARY_TYPE;
                                DICTIONARY_IN : out DICTIONARY_PTR;
                                FILENAME : in STRING) is

        begin
            return;
        end CREATE_DICTIONARY;

    procedure TOKEN_IS_FOUND(IN_DICTIONARY : out DICTIONARY_PTR;
                             WORD : in TOKEN_DEFINITION.TOKEN_TYPE;
                             FOUND : out BOOLEAN) is

        begin
            FOUND := TRUE;
        end TOKEN_IS_FOUND;

end DICTIONARY_MANAGER;

```

It is apparent that the procedure bodies are just stubs. References to these two procedures appear in procedures `STYLE_CHECKER` and `CHECK_OBJECT_NAMES_SIZE`. Upon closer examination of the package and these references, it can be determined that the dictionary manager package is for purposes of automatic spelling correction, a feature not currently present in the style checker. One of the references is seen in this code segment (from the latter procedure):

```

    DICTIONARY_MANAGER.TOKEN_IS_FOUND(STYLE_DICTIONARY,
        SPELL_CHECK_WORD,
        FOUND);
    if not FOUND then
        -- Not handled now...
        null;
    end if;

```

There are two straightforward solutions to this exercise. The first is to give values to the out mode parameters (which can be anything, since they are never used). This preserves the option of adding the spelling correction capability later. Appropriate comments in the code would be useful for subsequent maintainers.

The second solution is to remove all the related code. This is the more interesting solution because it forces the students to examine more code to be sure that they have found all the related code, and it also demands more thorough regression testing to determine that the existing functionality of the system has not been compromised.

4.8. Discrepancy Report 3: File Name Length Errors

Exercise

During the process of transporting the DASC system to the MS-DOS/Alsys Ada environment, all system files with names exceeding 8 characters were given modified names. Upon execution, the system could not find some files because they had different names. The system should be modified to work with the new 8-character names.

Information for Instructors

The objective of this exercise is to correct known errors in the system and to give the students an introduction to the Ada facilities for relating external and internal file names. This exercise is essential for students using MS-DOS; it probably can be ignored by other students.

The instructor may simply ask the students to find all occurrences of file names in the code and reduce them to 8 characters if necessary. The following are known occurrences and may be given to the students if desired.

- In the specification of package `FILE_HANDLING`:

```

HELP_FILE_NAME      : constant STRING := "STYLE_HELP.INI";
STYLE_DICTIONARY_NAME : constant STRING :=
    "STYLE_DICTIONARY.INI";

```

- In the body of package `FILE_HANDLING`:

```
COMMAND_LINE_FILE_NAME : constant STRING := "COMMANDLI.TXT";
```

4.9. Discrepancy Report 4: Empty Input File Error

Exercise

If the file input file (named `COMMANDLI.TXT` in the original version of the system) is empty or any line in that file is the name of a nonexistent file, several exceptions are raised and the DASC system fails to perform properly. Modify the system to provide better handling of these conditions.

Information for Instructors

The objective of this exercise is to let the students see the result of incompleteness in the specification. Of course, since the original requirements specification is not available, we cannot be sure that this is a specification error rather than a design and implementation error. However, such boundary condition errors are commonly overlooked in specifications, so it is likely to be such an error.

It is desirable that the system make a graceful exit if the list of files to process is empty, and that it simply report files that cannot be found and proceed to the next. The students should revise the requirements document to cover these cases. Then the code should be modified to reflect the new requirements. Regression testing should follow, and the test suite should be augmented to include the case of an empty `COMMANDLI.TXT` file.

4.10. Discrepancy Report 5: Unreachable Code

Exercise

The compiler reports unreachable code in function `IS_STATEMENT`. Determine the cause of this error and correct it.

Information for Instructors

The unreachable code is that shown below between the two `when` clauses:

```
case TOKENIZER.TYPE_OF_TOKEN_IS (EXAMINED_TOKEN) is
  ...
  when WHILE_TOKEN => return true;
    LOOKAHEAD := PREVIOUS_NON_TRIVIAL_TOKEN (EXAMINED_TOKEN);
    if TOKENIZER.TYPE_OF_TOKEN_IS
      (LOOKAHEAD) /= TOKENIZER.END_TOKEN then
```

```

        return true;
    else
        return false;
    end if;
    when ACCEPT_TOKEN => return true;
    ...
end case;

```

The instructor will need to identify the unreachable code unless the compiler does so. This discrepancy was noted using the VAX Ultrix/Verdix Ada environment, but not with the other environments in which the system was tested.

The unreachable code can simply be removed. The reason for its presence is unknown.

4.11. Code Reviews

Exercise

In conjunction with exercise 4.10, conduct a code inspection of procedure `is_statement`.

Information for Instructors

The objective of this exercise is to introduce students to the fundamentals of code reviews. Such reviews (walkthroughs and inspections) are among the most important tools available to software engineers for improving software quality. The instructor should require code reviews of all major changes to the code. (For more information on all kinds of software technical reviews see [Collofello88a] and [Cross88].)

To introduce the process, the students are asked to conduct a code review of an existing code module. Procedure `is_statement` has been chosen for two reasons. First, it is the subject of a discrepancy report (see exercise 4.10) and a code review is a good technique for determining how to remove this discrepancy.

Second, the procedure exhibits several occurrences of a common coding error and the review can be used to generate an additional change request to fix them. The error is a misuse of the boolean data type, as illustrated in this code segment from the procedure:

```

if TOKENIZER.TYPE_OF_TOKEN_IS
  (LOOKAHEAD) /= TOKENIZER.END_TOKEN then
  return true;
else
  return false;
end if;

```

The improved code is:

```

return TOKENIZER.TYPE_OF_TOKEN_IS (LOOKAHEAD) /= TOKENIZER.END_TOKEN;

```

4.12. Change Request 1: Improved Flaw and Style Messages

Exercise

Spelling errors have been noticed in the flaw and style reports generated by DASC; these are to be corrected. The errors are:

1. "Inconsistant Indentation" should be "Inconsistent Indentation"
2. "PRAGMA'S" and "PRAGMA's" should be "PRAGMAS"
3. "Reserve word ..." should be "Reserved word ..."
4. "upper case" should be "uppercase"; "lower case" should be "lowercase"

Information for Instructors

This is a very simple perfective maintenance exercise. Its objective is to let the students gain some familiarity with the packages that generate the reports. This familiarity will be useful for subsequent exercises.

Because the specific errors to be corrected are given, the students should be able to use the search capability of a text editor to find the occurrences of these errors. The only difficulty is identifying the packages and procedures to be searched. They are package `REPORT_GENERATOR` and procedure `RESERVE_WORD_ENCOUNTED`.

4.13. Change Request 2: User Interface, Version 1

Exercise

Currently the DASC system expects the list of file names to be processed to be in the file named `COMMANDLI.TXT`. Add a new user interface that, upon starting the system, prompts the user for a file name, reads in that file name, and then reads the names of files to be processed from that file.

Information for Instructors

The objective of this exercise is to introduce the students to writing entirely new code in response to a request for new functionality. This is the first of three exercises devoted to increased functionality of the user interface.

Note that the user interface supplied on the distribution diskette for the VAX VMS/DEC Ada environment is written in DEC Command Language (DCL), and is therefore not portable to any other environment. This and the next two exercises essentially provide the functionality of that user interface, but coded in Ada.

When doing these three exercises, the instructor may wish to assign them sequentially to introduce the students to the *iterative enhancement* technique for system building. Although originally intended for new development, this technique is equally applicable to maintenance when a significant amount of new code is being produced.

This exercise will allow students new to Ada to gain some familiarity with simple string input and output, and with making a correspondence between internal and external file names.

These exercises should include the writing of more detailed specifications and the modification of documentation as appropriate. Of particular interest is the modification of the test plan. The new user interface can only be tested interactively, so the test plan will need to contain a description of how this is to be done. The user manual will also require a substantial modification (see also exercise 4.4).

4.14. Change Request 3: User Interface, Version 2

Exercise

Modify the user interface of the previous exercise so that the user can build the file of file names interactively. The systems should repeatedly prompt the user for another file name, read the name, and append it to the file of file names. The user should be given a way to indicate that no more names are to be read, at which time the DASC system processes those files whose names have been read.

Information for Instructors

This exercise continues the development described for the previous exercise. The instructor should require the students to prepare a more detailed specification of the changes to be made and require appropriate changes to documentation, including the user manual. The section of the test plan describing interactive testing (see previous exercise) may need to be revised.

4.15. Change Request 4: User Interface, Version 3

Exercise

Modify the user interface to allow immediate screen display of flaw and style reports. After processing all the files whose names are in the input file (named `COMMANDLI.TXT` in the original version of the system), the system should ask the user if display of the flaw report is desired. If so, it is displayed on the screen one page at a time (like the UNIX or MS-DOS *more* command). After each page, the user can request another page or exit. A similar display of the style report should then be allowed. This is repeated for each file processed.

Information for Instructors

The objective of this exercise is to give the students an opportunity to add a significant new capability to the system. It will give the students a substantial amount of experience with interactive input and output in Ada.

As with the previous two exercises, the instructor should first require a detailed specification of the proposed change. This should include the wording of messages from the system and the responses that the user may give. The user manual describes this kind of interface, and it may be used as a starting point for the new specifications.

Documentation will again need to be modified. If the user manual was stripped of all references to the original VAX VMS user interface as a result of exercise 4.4, the students may now discover that they are putting almost all of the removed material back in. Therefore it is important for the instructor to structure the exercises so as not to frustrate the students unnecessarily. One approach is to do the three user interface modification exercises sequentially but with the understanding that the user manual will be modified only once, and that will be to reflect the final user interface. The actual modification of the user manual can *precede* the code changes, so that the manual can serve as a specification for the design of the new code.

4.16. Change Request 5: Add Page Headers to Reports

Exercise

Revise the format of flaw reports and style reports to include a page heading on each page. The heading should include the name of the Ada program file that generated the report, the date, and the report page number.

Information for Instructors

The objective of this exercise is to give the students experience reading and understanding an existing code module, and then making a relatively small change.

The package `REPORT_GENERATOR` contains the procedures that produce the reports. A recommended approach to a solution is to add a procedure that formats and prints a page heading, count the lines printed, and then invoke the page heading procedure at the appropriate points. Students might anticipate that different printer or display devices would have different numbers of lines per page, and so it is good practice to make the page size a named constant that can be easily changed.

4.17. Change Request 6: Add Line Numbers to Flaw Reports

Exercise

Revise the format of the flaw report so that the source file line number is reported for each line found to have a style flaw.

Information for Instructors

The objective of this exercise is also to give students experience in understanding a code module in order to make a simple change. The change needed is smaller than that of the previous exercise, but requires more program analysis.

The difficult task is understanding the organization of the program and the function of each of its components so that students can determine where to make the required modifications. As the style checker processes the code, it builds a linked list that it can then traverse in either direction. Because of this, source lines with flaws are sometimes reported out of order. Recognizing this behavior of the program is essential to finding a solution for this exercise.

An analysis of the source code reveals that the program already keeps track of the current line number in the variable `CURRENT_TOKEN.TOKEN_POSITION.LINE`. Since this variable is visible in the procedure `LINE_CONTAINING_TOKEN`, and this procedure is always used to produce the source line reported in the flaw report, one simple solution is to modify this procedure so that it concatenates the line number onto the beginning of the erroneous source line.

The instructor might expect the students to make the following errors on this exercise:

- It is easy to end up with a type mismatch when dealing with integers, strings, and dynamic strings all in the same statement.
- The code contains a number of variables that keep track of different but related counts, such as statement count, number of blank lines, and line within the file. Students could print out the wrong variable.

4.18. Change Request 7: Allow User-Specified Style Parameters

Exercise

Modify the system so that the quantifiable style parameters are read from a file rather than being directly coded into the system. This will allow different organizations to customize the system more easily.

Information for Instructors

The objective of this exercise is make a significant change to the system that will increase its usefulness. It requires that the students first understand the various style parameters and then decide which can meaningfully be customized. Then they must identify where those parameters appear in the code, change constants to variables, and provide a way for values of those variables to be read in.

The package `Style_Parameters`, and in particular, procedure `Set_Style_Parameters`, contains the code that defines the style parameters to be examined. The instructor should point out that the system is reasonably well designed in this respect; the students might be asked to imagine performing this exercise on a system in which these values are neither collected in a single package nor declared as named constants.

Regression testing after this modification should be planned carefully. A first round of tests should be performed with the style parameters in the input file being the same as those currently declared in the code. This will allow the new output to be compared to the known output from the test suite. Then the various parameters should be changed, the tests performed again, and the output examined. Many of the results will be different, and the students must determine manually if they are correct.

It will certainly be the case that for some values of some parameters, no program in the current test suite is an adequate test. New test program should be devised and placed in the test suite.

As with the previous changes, modifications of the documentation will be required. Students should not forget that the user manual will require revision as part of this exercise.

Annotated Bibliography

- [Britcher86] Britcher, Robert N., and James J. Craig. "Using Modern Design Practices to Upgrade Aging Software Systems." *IEEE Software* 3, 3 (May 1986), pp. 16-24.

This paper describes some software maintenance experiences within IBM.

- [Collofello88a] Collofello, James S. *The Software Technical Review Process*. Curriculum Module SEI-CM-3-1.5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.

Capsule Description: This module consists of a comprehensive examination of the technical review process in the software development and maintenance life cycle. Formal review methodologies are analyzed in detail from the perspective of the review participants, project management and software quality assurance. Sample review agendas are also presented for common types of reviews. The objective of the module is to provide the student with the information necessary to plan and execute highly efficient and cost effective technical reviews.

- [Collofello88b] Collofello, James S. *Introduction to Software Verification and Validation*. Curriculum Module SEI-CM-13-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.

Capsule Description: Software verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed. This curriculum module provides an overview needed to understand in-depth curriculum modules in the verification and validation area.

- [Conn87] Conn, Richard. *The Ada Software Repository and the Defense Data Network*. New York: New York Zoetrope, 1987.

This book describes the history and contents of the Ada Software Repository (the original source of the DASC system). It also describes several ways for interested persons to access the repository and extract software from it.

- [Cross88] Cross, John A., editor. *Support Materials for The Software Technical Review Process*. Support Materials SEI-SM-3-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1988.

This package includes a number of examples and structured exercises for students.

- [IEEE83] IEEE. *Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 829-1983, Institute of Electrical and Electronics Engineers, 1983.
- This book should be available for reference by all software engineering educators and students. Although some persons might disagree with some of the definitions, on the whole it is an excellent resource for those wishing to promote a standard vocabulary of software engineering.*
- [Parikh83] Parikh, Girish, and Nicholas Zvegintzov. *Tutorial on Software Maintenance*. IEEE Computer Society Press, 1983.
- This book is a collection of over thirty papers on software maintenance, mostly from the late 1970s and early 1980s. Much of the material is now dated, but on the whole it is some interesting background reading for the instructor.*
- [Swanson76] Swanson, E. B. "The Dimensions of Maintenance." *Proc. 2nd International Conference on Software Engineering*, IEEE Computer Society, 1976, pp. 492-497.
- This is one of the very early papers on software maintenance. It is still often cited for some of its definitions.*
- [Tomayko86] Tomayko, James E. *Support Materials for Software Configuration Management*. Support Materials SEI-SM-4-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1986.
- This package of support materials includes a number of examples of industrial discrepancy reports and change forms, an example of a configuration management plan, several kinds of classroom materials, and example of using a configuration management tool, and additional background material for instructors.*
- [Tomayko87a] Tomayko, James E. *Software Configuration Management*. Curriculum Module SEI-CM-4-1.3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1987.
- Capsule Description: Software configuration management encompasses the disciplines and techniques of initiating, evaluating, and controlling change to software products during and after the development process. It emphasizes the importance of configuration control in managing software production.*
- [Tomayko87b] Tomayko, James E. *Teaching a Project-Intensive Introduction to Software Engineering*. Technical Report CMU/SEI-87-TR-20, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., 1987.
- Abstract: This report is meant as a guide to the teacher of the introductory course in software engineering. It contains a case study of a course based on a large project. Additional materials used in*

teaching the course and samples of student-produced documentation are also available. Other models of course organization are also discussed.

Appendix 1. Project Team Roles

Principal Architect: Responsible for the creation of the software product. Primary responsibilities include authoring the requirements document and specification document, advising on overall design, and supervising implementation and testing.

Project Administrator: Responsible for resource allocation and tracking. Primary responsibilities are cost analysis and control, computer and human resource acquisition and supervision. Collects data and issues weekly cost/manpower consumption reports and the final report.

Configuration Manager: Responsible for change control. Primary responsibilities include writing the configuration management plan, tracking change requests and discrepancy reports, calling and conducting change control board meetings, archiving, and preparing product releases.

Quality Assurance Manager: Responsible for the overall quality of the released product. Primary responsibilities include preparing the quality assurance plan, calling and conducting reviews and code inspections, evaluating documents and tests.

Test and Evaluation Engineer: Responsible for testing and evaluating individual modules and subsystems and for preparing the appropriate test plans.

Designer: Primary responsibility is developing aspects of the design as specified by the architect. During the pre-design stage, this person could assist in a literature search to explore similar products or problems.

Implementor: Primary responsibility is to implement the individual modules of the design and serve as the technical specialist for a particular language and operating system. During the requirements specification and design stages, the implementors could develop tools and experiment with new language constructs expected to be needed in the product.

Documentation Specialist: Responsible for the appearance and clarity of all documentation and for the creation of user manuals.

Verification and Validation Engineer: Responsible for creating and executing test plans to verify and validate the software as it develops, including tracing requirements through specification, design, coding, and testing. Also responsible for code inspections. Acts as a member of an independent group.

Maintenance Engineer: Primary responsibility is creating a guide to the maintenance of the delivered product.

Note: The above definitions are reprinted from [Tomayko87b].

Appendix 2. Distribution Diskette Contents

Macintosh Version

The information on the source code diskette has been taken from the Ada Software Repository and is in the public domain. As a courtesy to the original developers of the system, it is requested that all copies of the software retain the prolog either as a separate file or as a prefix to the main program.

The information on the documentation diskette is copyright 1989 by Carnegie Mellon University. Permission to make copies or derivative works of this information is granted, without fee, provided that the copies and derivative works are not made or distributed for direct commercial advantage, and that all copies and derivative works cite this document by name and document number and give notice that the copying is by permission of Carnegie Mellon University.

Source Code Diskette

The diskette name is DASC Source, and it contains a copyright notice file and three folders named Source, Test Suite, and Tools (Figure 1). A typical folder display for each is shown below.

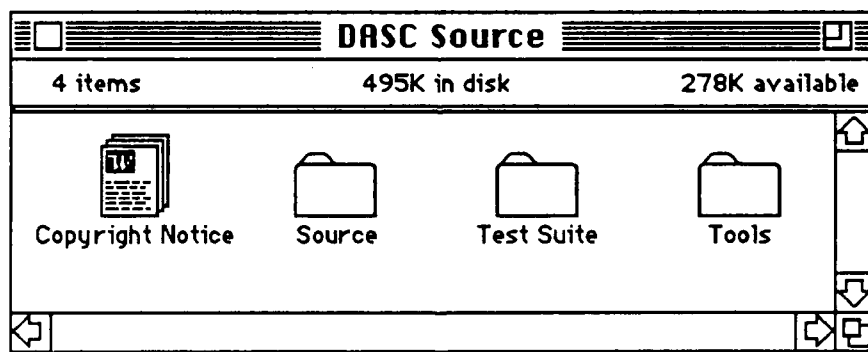
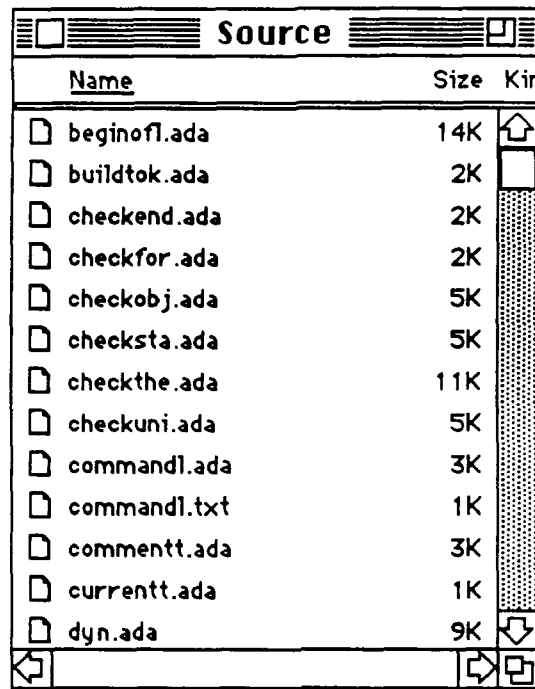


Figure 1. Contents of source code diskette

The source code folder (Figure 2) includes 63 Ada compilation units and two input files (command1.txt and styxhelp.ini) that are needed when the system is executed.



Name	Size	Kind
beginofl.ad	14K	File
buildtok.ad	2K	File
checkend.ad	2K	File
checkfor.ad	2K	File
checkobj.ad	5K	File
checksta.ad	5K	File
checkthe.ad	11K	File
checkuni.ad	5K	File
commandl.ad	3K	File
commandl.txt	1K	Text File
commentt.ad	3K	File
currentt.ad	1K	File
dyn.ad	9K	File

Figure 2. Source folder (partial listing)

The test suite folder (Figure 3) contains seven test files and two output files (test1.FLW and test1.STY) produced when the DASC system is run on file test1.ad.

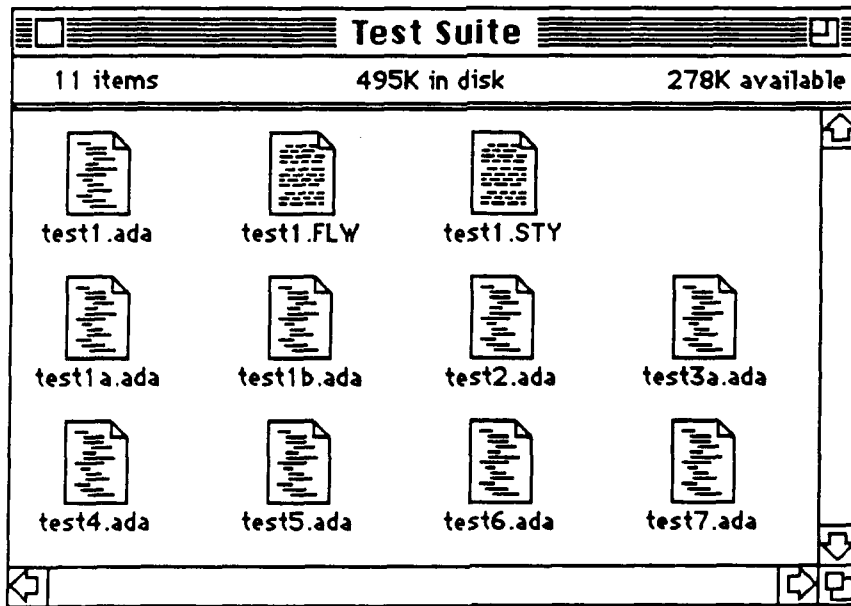


Figure 3. Test Suite folder

The tools folder (Figure 4) includes the following files:

install.doc	Describes how to install the version of DASC prepared by the IPFW students. It makes reference to diskettes used by the students; those references do not apply to the distribution diskettes that accompany this report.
dasc.com	A DEC Command Language (DCL) file that is the user interface to the DASC system on a VAX VMS computer system.
compile.doc	A listing of the order in which the 63 source files must be compiled.
dwn_vms.com	A DEC Command Language (DCL) file that changes the long file names used on a VMS system to the short (8 character) file names for an MS-DOS system.
up_vms.com	A DEC Command Language (DCL) file that changes the short (8 character) file names for an MS-DOS system to the long file names used on a VMS system.
dwn_unix.com	A UNIX C shell script that changes the long file names used on a UNIX system to the short (8 character) file names for an MS-DOS system.
up_unix.com	A UNIX C shell script that changes the short (8 character) file names for an MS-DOS system to the long file names used on a UNIX system.

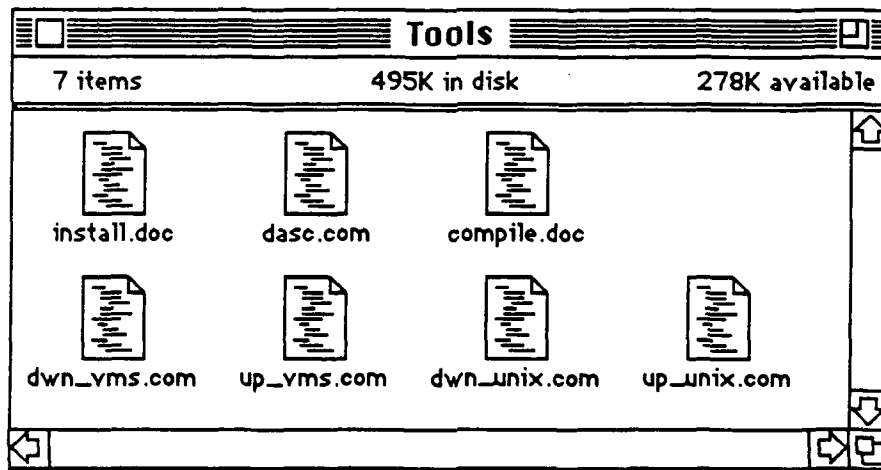


Figure 4. Tools folder

Documentation Diskette

The diskette name is DASC Doc, and it contains a copyright notice file and a folder named DASC Documentation. That folder contains three other folders, each of which contains a complete set of DASC documents in a different format (Microsoft Word, MacWrite, and text only). A typical folder display is shown in Figure 5.

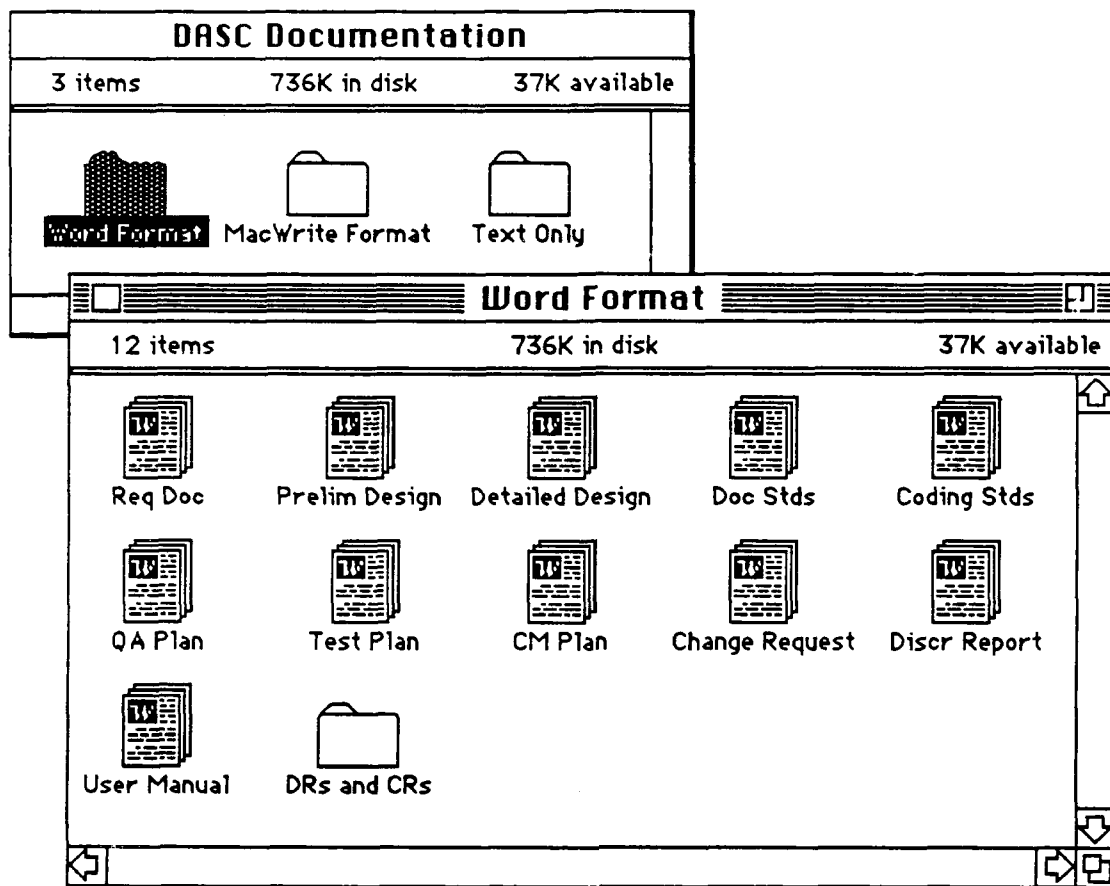


Figure 5. DASC Documentation and Word Format folders

The folder DRs and CRs contains the Microsoft Word versions of the discrepancy reports and change requests that appear as Attachment 1 of this document. These appear *only* in Microsoft Word format.

PC/AT Version

The diskette name is DASC, and it contains both the source code and the documentation. At the top level it contains a copyright notice file COPYRIGHT.TXT and four directories, named SOURCE, TSTSUIE, TOOLS, and DOCUMENT. Listings of the directories are shown below.

The information in directories SOURCE and TSTSUIE has been taken from the Ada Software Repository and is in the public domain. As a courtesy to the original developers of the system, it is requested that all copies of the software retain the prolog either as a separate file or as a prefix to the main program. The information in directory TOOLS is also in the public domain, either having come from the Ada Software Repository or having been placed in the public domain by Carnegie Mellon University.

The information in directory DOCUMENT is copyright 1989 by Carnegie Mellon University. Permission to make copies or derivative works of this information is granted, without fee, provided that the copies and derivative works are not made or distributed for direct commercial advantage, and that all copies and derivative works cite this document by name and document number and give notice that the copying is by permission of Carnegie Mellon University.

Top Level Directory Listing

Volume in drive A is DASC
Directory of A:\

```
SOURCE      <DIR>      2-09-89   2:18p
TSTSUIITE    <DIR>      2-09-89   2:18p
TOOLS        <DIR>      2-09-89   2:18p
DOCUMENT     <DIR>      2-09-89   2:18p
COPYRIGHT TXT 1014      2-09-89   1:01p
      5 File(s)      533504 bytes free
```

Source Directory Listing

Volume in drive A is DASC
Directory of A:\SOURCE

```
.          ..          BEGINOFL ADA  BUILDTOK ADA  CHECKEND ADA
CHECKFOR ADA  CHECKOBJ ADA  CHECKSTA ADA  CHECKTHE ADA  CHECKUNI ADA
COMMANDL ADA  COMMENTT ADA  CURRENTT ADA  DYN          ADA  ENTERB ADA
ENTERS ADA    EXITBL ADA  FILESPEC ADA  FIXIT ADA     GETNEXTT ADA
INSERT ADA    ISARESER ADA  ISSTATEM ADA  LINECONT ADA  LITERAL ADA
MANAGER ADA   NEWLINET ADA  NEXTCHAR ADA  NEXTIDEN ADA  NONTRIVI ADA
OBJECTNA ADA  REPGENBO ADA  REPGENSP ADA  RESERVDW ADA  RESERVEW ADA
SPARAMBO ADA  SPARAMSP ADA  SRCHBCKO ADA  SRCHBCKW ADA  SRCHFORE ADA
SRCHFORW ADA  STACKPAC ADA  STYLECHE ADA  TOKENDEF ADA  TOKENZBO ADA
TOKENZSP ADA  TREEROOT ADA  TYPEDECL ADA  HELPBODY ADA  HELPOISA ADA
HELPEXIT ADA  HELPFB ADA    HELPFIND ADA  HELPFS ADA    HELPGET ADA
HELPIB ADA    HELPINIT ADA  HELPIS ADA    HELPME ADA    HELPMENU ADA
HELPRESE ADA  HELPROMP ADA  HELPSPEC ADA  HELPTXT ADA   FILEBODY ADA
STYXHELP INI  COMMANDL TXT
      67 File(s)      533504 bytes free
```

Test Suite Directory Listing

Volume in drive A is DASC
Directory of A:\TSTSUIITE

```
.          ..          TEST1 ADA  TEST1A ADA  TEST1B ADA
TEST2 ADA    TEST3A ADA  TEST4 ADA  TEST5 ADA  TEST6 ADA
TEST7 ADA    TEST1 STY  TEST1 FLW
      13 File(s)      533504 bytes free
```

Tools Directory Listing

Volume in drive A is DASC
Directory of A:\TOOLS

.	..	DASC	COM	UP_UNIX	COM	UP_VMS	COM
DWN_UNIX	COM	DWN_VMS	COM	COMPILE	DOC	INSTALL	DOC

9 File(s) 533504 bytes free

The tools directory includes the following files:

dasc.com	A DEC Command Language (DCL) file that is the user interface to the DASC system on a VAX VMS computer system.
up_unix.com	A UNIX C shell script that changes the short (8 character) file names for an MS-DOS system to the long file names used on a UNIX system.
up_vms.com	A DEC Command Language (DCL) file that changes the short (8 character) file names for an MS-DOS system to the long file names used on a VMS system.
dwn_unix.com	A UNIX C shell script that changes the long file names used on a UNIX system to the short (8 character) file names for an MS-DOS system.
dwn_vms.com	A DEC Command Language (DCL) file that changes the long file names used on a VMS system to the short (8 character) file names for an MS-DOS system.
compile.doc	A listing of the order in which the 63 source files must be compiled.
install.doc	Describes how to install the version of DASC prepared by the IPFW students. It makes reference to diskettes used by the students; those references do not apply to the distribution diskettes that accompany this report.

Document Directory Listing

Volume in drive A is DASC
Directory of A:\DOCUMENT

.	..	CHNGREQ	TXT	CMPLAN	TXT	CODESTDS	TXT
DETDES	TXT	DISCRREP	TXT	DOCSTDS	TXT	PREDES	TXT
REQDOC	TXT	TESTPLAN	TXT	USERMAN	TXT	QAPLAN	TXT

13 File(s) 533504 bytes free

DASC DISCREPANCY REPORT

Report No.: 1

Release No.:

Originator:

Position:

E-Mail Address:

Date:

Problem Description/Requirement Not Met: During execution, an exception is raised
in the procedure ENTERING_BLOCK_STRUCTURE.

Correction Description:

Resource Estimation (Hrs)

Documentation Affected:

Modification

Testing

Other

TOTAL

Source File(s) Affected:

CCB Decision

Approved As Is ____ Waived ____ Approved For Analysis ____

Reasons Waived:

=====
CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC DISCREPANCY REPORT

Report No.: 2

Release No.:

Originator:
E-Mail Address:

Position:
Date:

Problem Description/Requirement Not Met: The compiler reports that *out* mode
parameters in two procedures are not given values. The errors are reported for
procedures **CREATE DICTIONARY** and **TOKEN IS FOUND**, both of which are
defined in package **DICTIONARY_MANAGER**.

Correction Description: _____

Resource Estimation (Hrs)

Modification _____

Testing _____

Other _____

TOTAL _____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is _____ Waived _____ Approved For Analysis _____

Reasons Waived: _____

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC DISCREPANCY REPORT

Report No.: 3
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Problem Description/Requirement Not Met: During the process of transporting the
DASC system to the MS-DOS/Alsys Ada environment, all system files with names
exceeding 8 characters were given modified names. Upon execution, the system
could not find some files, because they had different names.

Correction Description: _____

Resource Estimation (Hrs)

Documentation Affected:

Modification _____
Testing _____
Other _____
TOTAL _____

Source File(s) Affected:

CCB Decision

Approved As Is ____ Waived ____ Approved For Analysis ____
Reasons Waived:

=====

CCB Signatures:	Date:
-----------------	-------

Request Closed
Configuration Manager/Document Specialist Signature:

Date:

DASC DISCREPANCY REPORT

Report No.: 4
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Problem Description/Requirement Not Met: If the file input file (named
COMMANDLI.TXT in the original version of the system) is empty or any line in that
file is the name of a nonexistent file, several exceptions are raised and the DASC
system fails to perform properly.

Correction Description: _____

Resource Estimation (Hrs)

Modification	_____
Testing	_____
Other	_____
TOTAL	_____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ____ Waived ____ Approved For Analysis ____
Reasons Waived:

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC DISCREPANCY REPORT

Report No.: 5
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Problem Description/Requirement Not Met: The compiler reports unreachable code
in function IS_STATEMENT.

Correction Description: _____

Resource Estimation (Hrs)
Modification _____
Testing _____
Other _____
TOTAL _____

Documentation Affected:

Source File(s) Affected:

CCB Decision
Approved As Is ____ Waived ____ Approved For Analysis ____
Reasons Waived:

=====

CCB Signatures:	Date:
-----------------	-------

Request Closed
Configuration Manager/Document Specialist Signature:

Date:

DASC CHANGE REQUEST

Change Request No.: 1

Release No.:

Originator:
E-Mail Address:

Position:
Date:

Change Type

☒ New Feature ☐ Cost Reduction ☐ Other (describe)

Correction Description: Spelling errors have been noticed in the flaw and style
reports generated by DASC; these are to be corrected. The errors are:

1. "Inconsistant Indentation" should be "Inconsistent Indentation"
2. "PRAGMA'S" and "PRAGMA's" should be "PRAGMAS"
3. "Reserve word ..." should be "Reserved word ..."
4. "upper case" should be "uppercase"; "lower case" should be "lowercase"

Resource Estimation (Hrs)

Modification

Testing

Other

TOTAL

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ☐ Waived ☐ Approved With Modification ☐

Reasons Waived/Description of Modification:

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC CHANGE REQUEST

Change Request No.: 2

Release No.:

Originator:

Position:

E-Mail Address:

Date:

Change Type

☒ New Feature ☐ Cost Reduction ☐ Other (describe)

Correction Description: Currently the DASC system expects the list of file names to
be processed to be in the file named COMMANDLI.TXT. Add a new user interface that
upon starting the system, prompts the user for a file name, reads in that file name,
and then reads the names of files to be processed from that file.

Resource Estimation (Hrs)

Modification

Testing

Other

TOTAL

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ☐ Waived ☐ Approved With Modification ☐

Reasons Waived/Description of Modification:

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC CHANGE REQUEST

Change Request No.: 3
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Change Type
☒ New Feature ☐ Cost Reduction ☐ Other (describe)

Correction Description: Modify the user interface of the previous exercise so that the
user can build the file of file names interactively. The systems should repeatedly
prompt the user for another file name, read the name, and append it to the file of
file names. The user should be given a way to indicate that no more names are to
be read, at which time the DASC system processes those files whose names have
been read.

Resource Estimation (Hrs)

Modification	_____
Testing	_____
Other	_____
TOTAL	_____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ☐ Waived ☐ Approved With Modification ☐
Reasons Waived/Description of Modification:

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC CHANGE REQUEST

Change Request No.: 4
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Change Type

☒ New Feature ☐ Cost Reduction ☐ Other (describe)

Correction Description: Modify the user interface to allow immediate screen display
of flaw and style reports. After processing all the files whose names are in
COMMANDLI.TXT, the system should ask the user if display of the flaw report is
desired. If so, it is displayed on the screen one page at a time (like the UNIX or
MS-DOS more command). After each page, the user can request another page or
exit. A similar display of the style report should then be allowed. This is
repeated for each file processed.

Resource Estimation (Hrs)

Modification

Testing

Other

TOTAL

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ☐ Waived ☐ Approved With Modification ☐

Reasons Waived/Description of Modification:

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC CHANGE REQUEST

Change Request No.: 5
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Change Type
☒ New Feature ☐ Cost Reduction ☐ Other (describe)

Correction Description: Revise the format of flaw reports and style reports to include
a page heading on each page. The heading should include the name of the Ada
program file that generated the report, the date, and the report page number.

Resource Estimation (Hrs)

Modification _____
Testing _____
Other _____
TOTAL _____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ☐ Waived ☐ Approved With Modification ☐
Reasons Waived/Description of Modification:

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC CHANGE REQUEST

Change Request No.: 6
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Change Type

☒ New Feature ☐ Cost Reduction ☐ Other (describe)

Correction Description: Revise the format of the flaw report so that the source file
line number is reported for each line found to have a style flaw.

Resource Estimation (Hrs)

Modification _____
Testing _____
Other _____
TOTAL _____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ☐ Waived ☐ Approved With Modification ☐

Reasons Waived/Description of Modification:

=====
CCB Signatures:

=====
Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DASC CHANGE REQUEST

Change Request No.: 7
Release No.:

Originator:
E-Mail Address:

Position:
Date:

Change Type
☒ New Feature ☐ Cost Reduction ☐ Other (describe)

Correction Description: Modify the system so that the quantifiable style parameters
are read from a file rather than being directly coded into the system. This will
allow different organizations to customize the system more easily.

Resource Estimation (Hrs)

Modification _____
Testing _____
Other _____
TOTAL _____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ☐ Waived ☐ Approved With Modification ☐
Reasons Waived/Description of Modification:

=====

CCB Signatures:

Date:

Request Closed
Configuration Manager/Document Specialist Signature:

Date:

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

REQUIREMENTS DOCUMENT

**Req Doc
04/04/88
Version 1.0**

**Written By
Peggy Jones**

Table of Contents

1. Introduction.....	1
1.1. Problem Statement.....	1
1.2. Product Description.....	1
2. Product Components.....	1
2.1. Software.....	1
2.1.1. DASC Source Files.....	1
2.1.2. DASC Sample Files.....	1
2.2. Documentation.....	1
3. Functional Requirements.....	2
3.1. User Interface.....	2
3.1.1. Allow User Easy Access.....	2
3.1.2. Process Sample Input Files.....	2
3.1.3. Process User Input Files.....	2
3.1.4. Process Multiple Input Files.....	2
3.1.5. Access On-Line Help Facility.....	2
3.1.6. Keep User Appropriately Informed.....	2
3.1.7. Allow User Easy Exit.....	3
3.2. Style Definition.....	3
3.2.1. Range Parameters.....	3
3.2.1.1. Subprogram Size.....	3
3.2.1.2. Name Length.....	3
3.2.1.3. Code Section Size.....	3
3.2.1.4. Subprogram Parameters.....	3
3.2.1.5. Loop Exits.....	3
3.2.1.6. Line Length.....	3
3.2.1.7. Abbreviation Length.....	3
3.2.2. Percentage Parameters.....	3
3.2.2.1. Literal Usage.....	4
3.2.2.2. Universal Type Usage.....	4
3.2.3. Average Parameters.....	4
3.2.3.1. Comment Size.....	4
3.2.3.2. Name Length.....	4
3.2.3.3. Underscores.....	4
3.2.4. Optional Parameters.....	4
3.2.4.1. Use of Underscores.....	4
3.2.4.2. Consistent Indentation in Declarations.....	4
3.2.4.3. Consistent Indentation in Comments.....	4
3.2.4.4. Use of Blank Lines in Control Structures.....	4
3.2.4.5. Use of Loop Names.....	4
3.2.4.6. Use of Data Structures.....	4
3.2.4.7. Use of Attributes.....	4
3.2.4.8. Depth of Nesting.....	4
3.2.4.9. Use of Representation Specifications.....	4
3.2.4.10. Use of Address Clauses.....	4
3.2.5. Other Considerations.....	5
3.2.5.1. Pragmas.....	5
3.2.5.2. Host Dependent Packages.....	5
3.2.5.3. Character Set.....	5
3.2.5.4. Object Case.....	5

REQUIREMENTS DOCUMENT (V. 1.0)

3.2.5.5. Reserved Word Case.....	5
3.2.5.6. Reserved Word Usage.....	5
3.3. Style Reporting.....	5
3.3.1. Naming Conventions.....	5
3.3.1.1. Invalid Object Case.....	5
3.3.1.2. Invalid Keyword Case.....	5
3.3.1.3. Name Segment Size.....	5
3.3.1.4. Average Name Size.....	5
3.3.2. Physical Layout.....	5
3.3.2.1. Occurrences of Multiple Statements Per Line.....	5
3.3.2.2. Inconsistent Indentation.....	6
3.3.2.3. Missing Blank Lines.....	6
3.3.2.4. Loops Without Names.....	6
3.3.3. Information Hiding.....	6
3.3.3.1. Percent of Literal Usage.....	6
3.3.3.2. Percent of Universal Type Usage.....	6
3.3.3.3. Attribute Use.....	6
3.3.3.4. Ada-Specific Features Used.....	6
3.3.4. Modularity.....	6
3.3.4.1. Number of Subprogram Parameters.....	6
3.3.4.2. Subprogram Size.....	6
3.3.4.3. Number of Loop Exits.....	6
3.3.4.4. Nesting Levels.....	6
3.3.5. Comment Usage.....	6
3.3.5.1. Number of Comments.....	6
3.3.5.2. Average Comment Size.....	6
3.3.6. Transportability.....	6
3.3.6.1. Line Length Violations.....	6
3.3.6.2. Address Clauses Used.....	6
3.3.6.3. Pragmas Used.....	6
3.3.6.4. Representation Specifications Used.....	6
3.3.7. Keyword Usage.....	6
3.3.7.1. Keywords Used.....	7
3.3.7.2. Keywords Allowed.....	7
3.3.8. Error Listing.....	7
4. Physical Requirements.....	7
4.1. User Interface.....	7
4.2. Style Checking.....	7
5. Constraints.....	7
A. Revision History.....	7

1. Introduction

This Requirements Document defines the complete specifications of the technical requirements for the Documented Ada Style Checker (DASC) Project. This document is structured according to the DASC Documentation Standards and Guidelines.

1.1. Problem Statement

Objectively speaking, what is program style and how can it be measured? Program style has been defined as a "followed convention with respect to punctuation, capitalization, and typographical arrangement and display". There are tools to measure program efficiency, tools to measure program complexity, and tools to measure program cost. There is a need for a tool to measure program style.

1.2. Product Description

The DASC Project is a software tool that will quantitatively measure program style for Ada language programs. DASC will accept a syntactically correct input program, check that program against an established style convention, and output quantitative and objective evaluations of that input program.

2. Product Components

The DASC Project will include all implementation code files and all external documentation used in the design, production, testing, and maintenance of the project.

2.1. Software

The software components will include all source code files necessary to operate the DASC system as well as sample Ada input files.

2.1.1. DASC Source Files

The source code shall include all files that allow the user to perform the following functions:

1. Execute the system interface
2. Execute the system processing
3. Execute the system help facility

2.1.2. DASC Sample Files

DASC will provide at least ten sample input files. These files can be used as valid input and will include style errors and deficiencies that are detected by the Style Checker.

2.2. Documentation

The DASC Project shall be supported with appropriate external documentation. These documents shall be version numbered and updated as changes occur.

- | | |
|---|--|
| 1. Documentation Standards and Guidelines | - will describe the basic standards for text documents supporting the DASC Project |
| 2. Requirements Document | - will specify capabilities that the system must provide in order to solve the problem |
| 3. Configuration Management Plan | - will define all policies related to the management of change for the project |

REQUIREMENTS DOCUMENT (V. 1.0)

4. Quality Assurance Plan - will describe the standards and procedures to be implemented to ensure satisfactory confidence in the project
5. Coding Standards - will present guidelines which provide self-documentation, uniformity, and clarity to all project source code
6. Preliminary Design - will provide the overall preliminary Document architectural plan or structure of the project source code
7. Detailed Design Document - will provide the overall detailed architectural plan or structure of the project source code
8. Test Plan - will describe the objectives of testing, particular tools and techniques to be used, actual test cases, and expected results
9. User Manual - will detail all necessary information -- with examples -- needed by the user to effectively operate the system

3. Functional Requirements

Functional requirements for the DASC tool include requirements for a user interface, as well as for style definition and style reporting.

3.1. User Interface

DASC processing should be easily accomplished by even inexperienced users. Specifically, the user interface should perform the following functions.

3.1.1. Allow User Easy Access

DASC should allow the user easy access from a system prompt.

3.1.2. Process Sample Input Files

DASC should allow the user to process selected test input files.

3.1.3. Process User Input Files

DASC should allow the user to process his own input files.

3.1.4. Process Multiple Input Files

DASC should allow the user to process multiple input files without reinvoking the Style Checker.

3.1.5. Access On-Line Help Facility

DASC should allow the user to access an on-line help facility for assistance.

3.1.6. Keep User Appropriately Informed

DASC should provide messages to inform the user as to processing status.

3.1.7. Allow User Easy Exit

DASC should allow the user to easily exit the Style Checker and return to the system prompt.

3.2. Style Definition

REQUIREMENTS DOCUMENT (V. 1.0)

The DASC tool should be easily modifiable to check Ada code against the specific style used by an installation. A method to define the desired style should be provided. This method should provide easy access to specific parameters including ranges, percentages, averages, and optional checks. The values given these parameters will formally define the style convention.

3.2.1. Range Parameters

DASC should provide a method for setting minimum and or maximum values for the following parameters. Violations of these ranges may be flagged as style flaws.

3.2.1.1. Subprogram Size

Maximum and minimum size parameters (number of statements) for subprograms should be provided. Subprograms smaller than the minimum may be too small to measure in terms of style.

3.2.1.2. Name Length

A minimum length parameter should be provided for names. Names smaller than a certain number of characters may not effectively identify an object.

3.2.1.3. Code Section Size

A minimum number of lines parameter should be provided for code sections, such as control structures. Any code section below this minimum cannot be reasonably style checked.

3.2.1.4. Subprogram Parameters

Minimum and maximum parameters should be provided for the number of allowable subprogram parameters.

3.2.1.5. Loop Exits

A maximum parameter should be provided for the number of allowable loop exits.

3.2.1.6. Line Length

A maximum parameter should be provided for the length of a source code line.

3.2.1.7. Abbreviation Length

A minimum parameter should be provided for abbreviation lengths.

3.2.2. Percentage Parameters

DASC should provide a means of defining percentage measurements for the following parameters. Values falling outside defined percentages may be flagged as style flaws.

3.2.2.1. Literal Usage

A maximum allowable percentage of literal usage should be provided. This parameter would limit the use of literals in program bodies.

3.2.2.2. Universal Type Usage

A maximum allowable percentage should be provided. This parameter would discourage exclusive use of universal types versus user-defined types.

3.2.3. Average Parameters

REQUIREMENTS DOCUMENT (V. 1.0)

DASC should provide a method for defining average measurements for the following parameters. Values falling outside the averages may be flagged as style flaws.

3.2.3.1. Comment Size

A minimum average for comment size should be provided.

3.2.3.2. Name Length

A minimum average for name length should be provided.

3.2.3.3. Underscores

A minimum average for use of underscores should be provided.

3.2.4. Optional Parameters

DASC should allow the following parameters to be set **on** or **off**; **on** if the particular item is to be reported, **off** otherwise.

3.2.4.1. Use of Underscores

3.2.4.2. Consistent Indentation in Declarations

3.2.4.3. Consistent Indentation in Comments

3.2.4.4. Use of Blank Lines in Control Structures

3.2.4.5. Use of Loop Names

3.2.4.6. Use of Data Structures

3.2.4.7. Use of Attributes

3.2.4.8. Depth of Nesting

3.2.4.9. Use of Representation Specifications

3.2.4.10. Use of Address Clauses

3.2.5. Other Considerations

DASC should provide for the definition of the following style considerations.

3.2.5.1. Pragmas

DASC should allow the definition of which pragmas to flag. This shall include the choices: all pragmas; just system dependent pragmas; no pragmas.

3.2.5.2. Host Dependent Packages

DASC should allow the definition of packages that may be dependent on the host system. Use of these packages may be flagged.

3.2.5.3. Character Set

DASC should allow definition of the allowable character set for source code. Violations may be flagged.

3.2.5.4. Object Case

DASC should allow definition of the case to be used for object names.

3.2.5.5. Reserved Word Case

DASC should allow the definition of the case to be used for Ada reserved words.

3.2.5.6. Reserved Word Usage

DASC should allow the definition of limits on the use of Ada reserved words.

3.3. Style Reporting

Specified style problems should be summarized in report form. In addition, individual style errors should be noted. Style reports will be organized within the following categories.

3.3.1. Naming Conventions

The following statistics concerning names should be summarized.

3.3.1.1. Invalid Object Case

3.3.1.2. Invalid Keyword Case

3.3.1.3. Name Segment Size

3.3.1.4. Average Name Size

3.3.2. Physical Layout

The following statistics concerning physical layout should be summarized.

3.3.2.1. Occurrences of Multiple Statements Per Line

3.3.2.2. Inconsistent Indentation

3.3.2.3. Missing Blank Lines

3.3.2.4. Loops Without Names

3.3.3. Information Hiding

The following statistics concerning information hiding should be summarized.

3.3.3.1. Percent of Literal Usage

3.3.3.2. Percent of Universal Type Usage

3.3.3.3. Attribute Use

3.3.3.4. Ada-Specific Features Used

3.3.4. Modularity

The following statistics concerning program modularity should be summarized.

3.3.4.1. Number of Subprogram Parameters

3.3.4.2. Subprogram Size

3.3.4.3. Number of Loop Exits

3.3.4.4. Nesting Levels

3.3.5. Comment Usage

The following statistics concerning comment usage should be summarized.

3.3.5.1. Number of Comments

3.3.5.2. Average Comment Size

3.3.6. Transportability

The following statistics concerning program transportability should be summarized.

3.3.6.1. Line Length Violations

3.3.6.2. Address Clauses Used

3.3.6.3. Pragmas Used

3.3.6.4. Representation Specifications Used

3.3.7. Keyword Usage

The following statistics concerning the use of Ada keywords should be summarized.

3.3.7.1. Keywords Used

3.3.7.2. Keywords Allowed

3.3.8. Error Listing

Individual errors should be noted by listing the style flaw found, and listing the source code line in which the flaw appeared. DASC should provide a parameter to limit the number of duplicate errors listed.

4. Physical Requirements

Physical requirements of the DASC tool include requirements for a user interface and for style checking.

4.1. User Interface

The DASC user interface should be an interactive shell written in the host computer's command language. This shell should be menu driven with screen prompts and should echo all keyboard input.

4.2. Style Checking

DASC should input any syntactically correct Ada compilation unit. These units will be checked against the user-defined style convention. Output will consist of a statistical summary of style considerations and individual style flaws. This output should be available for screen viewing and/or printing.

5. Constraints

REQUIREMENTS DOCUMENT (V. 1.0)

The Documented Ada Style Checker (DASC) Project will be developed on Digital Equipment Corporation (DEC) using the Virtual Address eXtended (VAX) architecture, Virtual Memory System (VMS) operating system. The compiler used for the project will be the DEC Ada compiler, version 1.4. Status prompts to the screen will be provided when any processing function shall exceed five seconds.

A. Revision History

Version 0.1	02/09/88	Peggy Jones
Revision 0.2	03/04/88	Peggy Jones Restructured the document according to the DASC Documentation Standards and Guidelines. The availability of the help facility was included in the document.
Revision 0.3	03/24/88	Peggy Jones Categorized the style definition (section 3.2), added functional requirements for the user interface (section 3.1), and included a Revision History section (section A).
Baselined	04/04/88	Bill Davis

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

PRELIMINARY DESIGN DOCUMENT

**Prelim Design
05/02/88
Version 1.0**

**Written By
Robin Mitchell**

Table of Contents

1. Introduction.....	1
1.1. Inheritance Charts.....	1
1.2. Descriptions.....	1
2. Reports.....	1

1. Introduction

This is the preliminary design document for the Documented Ada Style Checker (DASC) project. This document provides the overall architectural design for the DASC project. Tools used to produce this document are the following:

- Excelerator
- Compilation Order tool

Contained in this document are the following:

- Inheritance charts for each package of the DASC project
- A brief description of each package along with what it exports

[Editor's note: The inheritance charts and package descriptions are not available for distribution with this document.]

1.1. Inheritance Charts

The inheritance charts consist of labeled square blocks with an arrow to indicate inheritance. The label for each package consists of a number, in the format $n.m$ followed by the package name. In this label n is the lowest vertical level on which the package appears, and m is a sequential numbering of the packages on that level.

1.2. Descriptions

The descriptions of the DASC system packages were generated as Excelerator reports. These descriptions have the following information:

Label	This is the same label as described in section 1.1 Inheritance charts.
Explodes	This is not applicable to the report and should be blank.
Location	This is not applicable to the report and should contain N/A.
Type	This is used to indicate the type of object being described and should contain the keyword PACKAGE.
Value	This is not applicable to the report and should contain N/A.
Description	This is used to describe the package and should contain a brief description along with a list of the entities the package exports.

All of the information after the description is not applicable and may be ignored.

2. Reports

The inheritance charts and descriptions follow.

[Editor's note: The inheritance charts and package descriptions are not available for distribution with this document.]

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

DETAILED DESIGN DOCUMENT

**Detailed Design
05/02/88
Version 1.0**

**Written By
Robin Mitchell**

Table of Contents

1. Introduction.....1

 1.1. Invoked-By List.....1

 1.2. Structure Charts.....1

2. Reports.....1

1. Introduction

This is the detailed design document for the Documented Ada Style Checker (DASC) project. This document provides the internal design for the DASC project. Tools used to produce this document are the following:

- Excelerator
- Compilation Order tool

Contained in this document are the following:

- An invoked-by list for each procedure in the DASC project
- Structure charts for each procedure in the DASC project

[Editor's note: The structure charts are not available for distribution with this document.]

1.1. Invoked-By List

The invoked-by list contains all the the procedures and functions ordered by number. Each number is of the format *n.m.o*, where *n.m* is the package number in which the procedure or function is contained, and *o* is a sequential numbering of the procedures and functions contained in that package. For an explanation of the package numbering system see the Preliminary Design Document. Under each procedure or function is a list of all the procedures and functions that invoke it. For example, if 1.0.1 check the style is the procedure being considered, it would appear as follows:

1.0.1 check the style
1.0.0 style checker

In this example 1.0.1 check the style is invoked only by 1.0.0 style checker.

1.2. Structure Charts

The structure charts depict a single level of decomposition for each procedure and function. A single level of decomposition was decided on due to size constraints of Excelerator and the printer used. Each procedure and function is labeled by a number and the procedure or function name.

2. Reports

The invoked-by list and structure charts follow:

1.0.0 style checker
this is the main procedure and is invoked at run-time

1.0.1 check the style
1.0.0 style checker

1.0.2 check object names size
1.0.1 check the style

1.0.3 check for attribute
1.0.1 check the style

1.0.4 check universal
1.0.1 check the style

1.0.5 get next token and update count
1.0.1 check the style

- 1.0.6 is statement
 - 1.0.1 check the style
 - 1.0.7 beginning of line indentation
 - 1.0.8 check statements per line
 - 1.0.9 reserved word encountered
 - 1.0.27 entering block structure
- 1.0.7 beginning of line indentation
 - 1.0.1 check the style
- 1.0.8 check statements per line
 - 1.0.1 check the style
- 1.0.9 reserved word encountered
 - 1.0.1 check the style
- 1.0.10 new line token encountered
 - 1.0.1 check the style
- 1.0.11 object name encountered
 - 1.0.1 check the style
- 1.0.12 literal encountered
 - 1.0.1 check the style
- 1.0.13 comment token encountered
 - 1.0.1 check the style
- 1.0.14 check for end of blocks
 - 1.0.1 check the style
- 1.0.15 tree follower
 - 1.0.2 check object names size
 - 1.0.15 tree follower (recursive)
- 1.0.16 next non trivial token
 - 1.0.3 check for attribute
 - 1.0.4 check universal
 - 1.0.9 reserved word encountered
 - 1.0.19 search forward for one of
 - 1.0.24 package token handler
 - 1.0.25 task token handler
 - 1.0.26 function procedure token handler
- 1.0.17 previous non trivial token
 - 1.0.6 is statement
 - 1.0.7 beginning of line indentation
 - 1.0.8 check statements per line
 - 1.0.9 reserved word encountered
 - 1.0.22 loop continuation
 - 1.0.26 function procedure token handler
 - 1.0.27 entering block structure
 - 1.0.30 search backward for one of
- 1.0.18 is universal
 - 1.0.4 check universal
- 1.0.19 search forward for one of

DETAILED DESIGN DOCUMENT (V. 1.0)

- 1.0.4 check universal
- 1.0.9 reserved word encountered
- 1.0.20 other indent cases
 - 1.0.7 beginning of line indentation
- 1.0.21 is loop name
 - 1.0.7 beginning of line indentation
- 1.0.22 loop continuation
 - 1.0.7 beginning of line indentation
- 1.0.23 match paren
 - 1.0.9 reserved word encountered
 - 1.0.23 match paren (recursive)
 - 1.0.26 function procedure token handler
- 1.0.24 package token handler
 - 1.0.9 reserved word encountered
- 1.0.25 task token handler
 - 1.0.9 reserved word encountered
- 1.0.26 function procedure token handler
 - 1.0.9 reserved word encountered
- 1.0.27 entering block structure
 - 1.0.9 reserved word encountered
 - 1.0.24 package token handler
 - 1.0.25 task token handler
 - 1.0.26 function procedure token handler
- 1.0.28 entering sub block structure
 - 1.0.9 reserved word encountered
- 1.0.29 caseless char
 - 1.0.9 reserved word encountered
 - 1.0.11 object name encountered
- 1.0.30 search backward for one of
 - 1.0.9 reserved word encountered
 - 1.0.27 entering block structure
- 1.0.31 handle parameter list
 - 1.0.27 entering block structure
- 1.0.32 search backward
 - 1.0.27 entering block structure
- 2.0.0 character set
 - 1.0.1 check the style
- 2.0.1 is declaration indentation required
 - 1.0.7 beginning of line indentation
- 2.0.2 small word size
 - 1.0.7 beginning of line indentation
 - 1.0.15 tree follower

DETAILED DESIGN DOCUMENT (V. 1.0)

- 2.0.3 vowel frequency
 - 1.0.2 check object names size
- 2.0.4 address clause allowed
 - 1.0.9 reserved word encountered
- 2.0.5 representation spec allowed
 - 1.0.9 reserved word encountered
- 2.0.6 is a predefined pragma
 - 1.0.9 reserved word encountered
- 2.0.7 note pragmas
 - 1.0.9 reserved word encountered
- 2.0.8 is a proscribed package
 - 1.0.9 reserved word encountered
- 2.0.9 line size
 - 1.0.10 new line token encountered
- 2.0.10 number of errors to report
 - 2.4.4 put flaw
- 2.0.11 is comment indentation required
 - 1.0.13 comment token encountered
 - 1.0.27 entering block structure
- 2.0.12 subprogram nesting level
 - 1.0.27 entering block structure
- 2.1.0 pop
 - 1.0.1 check the style
 - 1.0.14 check for end of blocks
 - 1.0.27 entering block structure
 - 1.0.28 entering sub block structure
- 2.1.1 push
 - 1.0.1 check the style
 - 1.0.9 reserved word encountered
 - 1.0.14 check for end of blocks
 - 1.0.27 entering block structure
 - 1.0.28 entering sub block structure
- 2.1.2 stack underflow
 - 1.0.27 entering block structure
- 2.3.0 token is found
 - 1.0.15 tree follower
- 2.3.1 create dictionary
 - 1.0.0 style checker
- 2.4.1 generate report
 - 1.0.0 style checker
- 2.4.2 put report line

DETAILED DESIGN DOCUMENT (V. 1.0)

- 2.4.1 generate report
- 2.4.3 print list
 - 2.4.1 generate report
- 2.4.4 put flaw
 - 1.0.1 check the style
 - 1.0.4 check universal
 - 1.0.7 beginning of line indentation
 - 1.0.8 check statements per line
 - 1.0.9 reserved word encountered
 - 1.0.10 new line token encountered
 - 1.0.11 object name encountered
 - 1.0.14 check for end of blocks
 - 1.0.27 entering block structure
- 2.4.5 insert into list
 - 1.0.9 reserved word encountered
- 3.0.0 build tokens
 - 1.0.0 style checker
- 3.0.1 next character
 - 3.0.0 build tokens
 - 3.0.12 next identifier
- 3.0.2 type of token is
 - 1.0.8 check statements per line
 - 1.0.19 search forward for one of
 - 1.0.23 match paren
 - 1.0.24 package token handler
 - 1.0.25 task token handler
 - 1.0.26 function procedure token handler
 - 1.0.27 entering block structure
 - 1.0.30 search backward for one of
- 3.0.3 insert
 - 3.0.0 build tokens
- 3.0.4 previous token
 - 1.0.1 check the style
 - 1.0.13 comment token encountered
- 3.0.5 type of token is
 - 1.0.1 check the style
 - 1.0.3 check for attribute
 - 1.0.4 check universal
 - 1.0.6 is statement
 - 1.0.16 next non trivial token
 - 1.0.17 previous non trivial token
 - 1.0.20 other indent cases
 - 1.0.21 is loop name
 - 1.0.22 loop continuation
 - 3.0.0 build tokens
- 3.0.6 external representation
 - 1.0.1 check the style

DETAILED DESIGN DOCUMENT (V. 1.0)

- 1.0.9 reserved word encountered
- 1.0.11 object name encountered
- 1.0.18 is universal
- 3.0.7 get line
 - 3.0.1 next character
- 3.0.8 next token
 - 1.0.5 get next token and update count
 - 1.0.8 check statements per line
 - 1.0.16 next non trivial token
 - 1.0.17 previous non trivial token
- 3.0.9 token position
 - 1.0.5 get next token and update count
 - 1.0.7 beginning of line indentation
 - 1.0.9 reserved word encountered
 - 1.0.10 new line token encountered
 - 1.0.13 comment token encountered
 - 1.0.30 search backward for one of
- 3.0.10 line containing token
 - 1.0.7 beginning of line indentation
 - 1.0.19 search forward for one of
 - 1.0.23 match paren
 - 1.0.27 entering block structure
 - 1.0.30 search backward for one of
- 3.0.11 length of comment
 - 1.0.13 comment token encountered
- 3.0.12 next identifier
 - 3.0.0 build tokens
- 3.0.13 look ahead
 - 3.0.12 next identifier
- 3.0.14 push back character
 - 3.0.12 next identifier
- 3.0.15 is a reserved word
 - 3.0.12 next identifier
- 3.0.16 reserved word
 - 3.0.12 next identifier
- 4.0.0 input file id
 - 3.0.0 build tokens
- 4.0.1 output file id
 - 1.0.0 style checker
- 4.1.0 str
 - 1.0.1 check the style
 - 1.0.9 reserved word encountered
 - 1.0.11 object name encountered
 - 1.0.15 tree follower
 - 1.0.30 search backward for one of

DETAILED DESIGN DOCUMENT (V. 1.0)

4.1.1 length

- 1.0.1 check the style
- 1.0.9 reserved word encountered
- 1.0.11 object name encountered
- 1.0.15 tree follower

4.1.2 upper case

- 1.0.18 is universal
- 2.4.5 insert into list
- 3.0.3 insert

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

DOCUMENTATION STANDARDS AND GUIDELINES

**Doc Stds
02/18/88
Version 2.2**

**Written By
Bill Davis**

Table of Contents

1. Introduction.....	1
2. Document Naming.....	1
3. Document Content.....	1
4. Document Appearance.....	1
4.1. Title Page.....	1
4.1.1. Primary Title.....	1
4.1.2. Secondary Title.....	1
4.1.3. File Name.....	1
4.1.4. Document Date.....	1
4.1.5. Version Number.....	2
4.1.6. Written By.....	2
4.1.7. Author(s) Name(s).....	2
4.2. Table Of Contents Page.....	2
4.2.1. Secondary Title.....	2
4.2.2. Table Of Contents.....	2
4.2.3. Contents Section.....	2
4.2.4. Section Page.....	2
4.2.5. Page Numbers.....	2
4.3. Document Body.....	2
4.4. Section Headings.....	3
4.5. Section Numbering.....	3
5. Acronym Usage.....	3
6. Project Titles.....	3
7. Revision History.....	4
A. Revision History.....	4

1. Introduction

This attachment describes the basic standards for text documents supporting the Documented Ada Style Checker (DASC) Project. These guidelines govern the appearance of all text documents. This attachment itself is an example of these standards.

2. Document Naming

ALL documents must be given the primary title DOCUMENTED ADA STYLECHECKER (DASC) PROJECT. The secondary title will be a simple, unique name, preferably a few words in length. This secondary title should allow the reader to determine the major scope of the particular document.

3. Document Content

Each document shall contain a title page, a table of contents, and a document body. The document body shall be the exhaustive authority on the DASC project for the particular subject therein. Only non-essential references to other documents may be mentioned. The authors of each document must ensure that all information concerning the document subject is clearly detailed in the document itself. In addition, each document shall contain a revision history. This history is more specifically detailed later in this document.

4. Document Appearance

All text documents shall follow these guidelines to ensure uniform appearance. All references to font, style, and character size (x-point) correspond to the *Ready-Set-Go* software package used on the Macintosh Plus Personal Computer. The font (print type) used with all text documents shall be New Century Schoolbook font.

4.1. Title Page

The first page of the document is the title page and will consist of seven parts.

4.1.1. Primary Title

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT, bold style in 24-point with the first letters that form the acronym (DASC) in 30-point size, centered in the upper third of the page, ALL uppercase letters will be used.

4.1.2. Secondary Title

Short, unique name describing the document content, bold style in 24-point, centered in the middle of the page, ALL uppercase letters will be used.

4.1.3. File Name

The name of the document on the accompanying floppy disc, bold style in 14-point, left-justified in the lower third of the page, first letter of filename will be in uppercase.

4.1.4. Document Date

The date MM/DD/YY the document was last modified, bold style in 14-point, left-justified immediately below the filename.

4.1.5. Version Number

The word 'Version', bold style in 14-point with the number of the form m.n, where m is the major version number and n the minor version number, left-justified, immediately below the document date.

4.1.6. Written By

Words 'Written By', Bold style in 14-point, beginning directly below the word **CHECKER** of primary title, on the same line as the filename.

4.1.7. Author(s) Name(s)

Name(s) of the document author(s), bold style in 14-point, located directly below words 'Written By', first letter of each name in uppercase.

4.2. Table Of Contents Page

All documents will contain a table of contents comprised of five parts.

4.2.1. Secondary Title

The secondary title of the document, followed by (V. m.n) where m is the major and n the minor version number, bold style in 10-point, one half inch from top of the page and centered on the page, ALL characters in uppercase.

4.2.2. Table Of Contents

Words 'Table Of Contents', Bold style in 14-point, centered on the page, one inch below the auxiliary title, ALL characters in uppercase.

4.2.3. Contents Section

Section number and name as they appear in the document, bold style in 14-point, left-justified with the first letter of each word in uppercase, indented five spaces for each level of nesting, with a maximum of three levels.

4.2.4. Section Page

The page number will correspond to the content section, bold style in 12-point, the page on which the section begins will be used, placed one inch from the right margin on the same line as the associated section name.

4.2.5. Page Numbers

The page number of the table of contents, bold style in 12-point, the pages of the table of contents should be numbered with lowercase Roman numerals (i, ii, iii, iv) at the outer margin on the bottom of the page.

4.3. Document Body

The document body will be produced in single column format. Each page of the body will contain the secondary title centered at the top of the page in bold, uppercase 10-point lettering. Each page will be successively numbered beginning with a single '1' on the first page, placed at the outer margin on the bottom of the page. Normal uppercase and lowercase rules will apply to the text body. Ten point, plain style will be the standard print size. Bold and/or uppercase will be used for emphasis.

4.4. Section Headings

The beginning of each section and subsection should be a single line consisting of the section number followed by the section name. Section headings will be bold style, 14-point, with the first letter of each word in uppercase. The headings will accurately describe the material in the section. Topic headings will be indented five spaces for each level of nesting.

4.5. Section Numbering

All sections of the document should be numbered to facilitate referencing. Section numbers are comprised of integers and decimal points beginning with 1.0. Individual sections may be subsectioned, but a single subsection is not permitted. Sections may be subsectioned as follows:

1. First Major Section
 - 1.1. First Minor Section
 - 1.2. Second Minor Section
 - 1.2.1. First Minor Major Section
 - 1.2.2. Second Minor Major Section
 - 1.2.2.1. First Minor Major Minor Section
 - 1.2.2.2. Second Minor Major Minor Section
- 2.0 Second Major Section

Sections may be subsectioned to as many levels as is necessary to describe their content.

5. Acronym Usage

An acronym is a word formed from the first letter(s) of major words of a compound term or phrase. Documents contained in the Documented Ada Style Checker (DASC) Project will make extensive use of acronyms. However, extreme care will be exercised by the authors of each document. The first use of an acronym in each document will require the full expansion of the acronym. After initial expansion, the acronym may be used as a substitute for the term/phrase. In lengthy documents, periodic expansions of acronyms should be accomplished no less than every other page. These periodic acronym expansions will occur with the initial use on a particular page. The method for expansion shall be as follows:

- The term/phrase fully spelled with the significant letter(s) boldfaced
- The acronym will follow the term/phrase enclosed in parentheses
- The acronym (with its parentheses) shall be boldfaced. The above method will alert readers to the acronyms and hopefully, help them to better remember their meaning.

The refresher expansions will serve to reinforce the acronym meaning for each reader.

6. Project Titles

There are ten position titles associated with the Documented Ada Style Checker (DASC) Project. No position title may be referenced by mere letters in any document. However, the letters CM may be used to reference the function of Configuration Management (CM) and likewise QA to reference the function of Quality Assurance (QA). No other titles may be shortened. Observe the acronym usage rule (section 5.0 in this document) when initially using Configuration Management (CM) or Quality Assurance (QA) functions as acronyms. The following is a listing of official Project titles:

- Project Manager
- Project Consultant
- Principal Architect
- Project Administrator
- Configuration Management (CM) Manager
- Quality Assurance (QA) Manager
- Project Tester
- Project Designer
- Project Implementer
- Document Specialist

7. Revision History

Each document shall contain a separate history section. The section shall be titled "A. Revision History" and shall be the last section of each document. The following information shall be listed for each version of the document:

DOCUMENTATION STANDARDS AND GUIDELINES (V. 2.2)

- The version number
- The date the revision was made
- The author(s) of the revision
- A brief outline of the revision

The document author is responsible for the revision history prior to document approval at the formal inspection review. The Document Specialist is fully responsible for the revision history after the initial, formal approval (baselining) of the document.

A. Revision History

Baselined	01/28/88	Bill Davis
Revision 2.0	02/09/88	Bill Davis Added Revision History material to original document. Sections amended include 3.0, 4.0, 4.1, 5, and A.
Revision 2.1	02/16/88	Bill Davis Added section 5.0 Acronym Usage and section 6.0 Project Titles to document.
Revision 2.2	02/18/88	Bill Davis Clarified responsibilities for revision history in section 7.0.

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

CODING STANDARDS

**Coding Stds
04/04/88
Version 1.0**

**Written By
Kathy Hoang
Janet Schnerre**

Table of Contents

1. Introduction.....	1
2. Overview.....	1
3. Error Log File.....	1
4. Heading Documentation.....	1
4.1. File Headings for Modifications and Error-repair.....	2
4.2. Procedure or Function Subprogram Headings.....	2
4.3. Package Specification Headings.....	3
4.4. Package Body Headings.....	3
5. In-line Documentation.....	4
A. Revision History.....	5

1. Introduction

This document presents a set of guidelines which provide self documentation, uniformity and clarity to all Documented Ada Style Checker (DASC) source code. These guidelines pertain to the documentation of the changes to the source code. Changes to the source code will match the code's existing style whenever possible.

2. Overview

A working version of DASC exists, and yet the code is likely to undergo substantial change during its useful life cycle. There are two primary reasons for making changes to the source code of an existing production program.

1. Fix an error in DASC. An error occurs when the source code does not perform the intended function. Each error correction must be documented in an error log file, in the file heading of the file that was changed, in the subprogram or package heading, and in the code itself.
2. Modify the source code. A modification changes the function of the code. Each modification must be documented in the file heading, the subprogram or package heading, and in the source code. Detailed explanation of the contents of the documentation, along with examples, is in the following sections.

3. Error Log File

The error log file must be maintained by the Project Implementer. This file will serve as a source of warning against errors for future implementers of DASC. Each time it is necessary to change the source code so that it functions correctly, an entry must be made in the error log file. The entry must contain the following information:

- Discrepancy Report number
- Project Implementer and date
- File and subprogram or package name in which error was located
- How error was discovered
- Changes made to repair the error

Each entry in the error log must be in the following format:

DISCREPANCY REPORT NUMBER:
PROJECT IMPLEMENTER:
DATE:

LOCATION OF ERROR: Give the filename and specification or body that contains the error.

DESCRIPTION OF DISCOVERY OF THE ERROR: Explain the circumstances in which the error was discovered.

CHANGES MADE TO REPAIR THE ERROR: Briefly describe the changes made to the source code to correct the error.

4. Heading Documentation

Heading documentation introduces the file, subprogram or package by supplying the reader with historical information.

4.1. File Headings for Modifications and Error-repair

Information describing the change must be added following the existing file heading. This must include the following:

- Release number
- Change Request or Discrepancy Report number
- Project name, Project Implementer and date
- Purpose of modification
- Procedure, function, or package that was modified

The heading documentation must be in the following format:

```
-- RELEASE NO:
-- CHANGE REQUEST/DISCREPANCY REPORT NO:
-- PROJECT NAME:
-- PROJECT IMPLEMENTER:
-- DATE:
--
-- MODIFICATION PURPOSE: Briefly describe the purpose for the modification.
--
-- PROCEDURES MODIFIED: List the procedures that were modified within main.
--
-- FUNCTIONS MODIFIED: List the functions that were modified within main.
--
-- PACKAGES MODIFIED: List the packages that were modified.
```

Further information must be added to the heading for the procedure, function or package that is modified.

4.2. Procedure or Function Subprogram Headings

Information describing the change to the subprogram must be added following the existing subprogram heading. This must include the following:

- Change Request or Discrepancy Report number
- Project Implementer and the date
- Description of change
- Changes to the procedure's parameters or algorithms
- Changes in the list of subprograms that the procedure invokes

The heading documentation must be in the following format:

```
-- CHANGE REQUEST/DISCREPANCY REPORT NO:
-- PROJECT IMPLEMENTER:
-- DATE:
--
-- DESCRIPTION OF CHANGE: Briefly describe all changes made in the source code including how
-- it worked originally and how it works after the modification.
--
-- PARAMETERS CHANGED: List all changes made to the original parameters (if any).
--
-- ALGORITHMS CHANGED: Describe the new algorithm if the old algorithm no longer applies to
-- the code.
--
-- SUBPROGRAMS MODIFIED: List the invoked subprograms that were changed.
```

4.3. Package Specification Headings

Information describing the change must be added following the existing file heading. This must include the following:

- Change Request or Discrepancy Report number
- Project Implementer and the date
- Description of change
- Changes in the resources used

The heading documentation must be in the following format:

```
-- CHANGE REQUEST/DISCREPANCY REPORT NO:
-- PROJECT IMPLEMENTER:
-- DATE:
--
-- DESCRIPTION OF CHANGE: Briefly describe all changes made in the source code including how
-- it worked originally and how it works after the modification.
--
-- RESOURCES MODIFIED: List changes in the use of resources from imported packages. This
-- includes types, constants, variables, procedures, functions and exceptions.
```

If a procedure within the package specification is modified, the following information must appear after the existing procedure heading:

- Change Request or Discrepancy Report number
- Project Implementer and the date
- Description of change
- Changes in the exceptions raised by the procedure

The heading documentation must be in the following format:

```
-- CHANGE REQUEST/DISCREPANCY REPORT NO:
-- PROJECT IMPLEMENTER:
-- DATE:
--
-- DESCRIPTION OF CHANGE: Briefly describe all changes made in the source code including how
-- it worked originally and how it works after the modification.
--
-- EXCEPTIONS RAISED: List the changes in the exceptions raised by the procedure.
```

4.4. Package Body Headings

Information describing the change must be added following the existing file heading. This must include the following:

- Change Request or Discrepancy Report number
- Project Implementer and the date
- Description of change
- Changes in the resources used

The heading documentation must follow this format:

```
-- CHANGE REQUEST/DISCREPANCY REPORT NO:
-- PROJECT IMPLEMENTER:
-- DATE:
--
-- DESCRIPTION OF CHANGE: Briefly describe all changes made in the source code including how
-- it worked originally and how it works after the modification.
--
-- RESOURCES MODIFIED: List changes in the use of resources from exported packages. This
-- includes types, constants, variables, procedures, functions and exceptions.
```


CODING STANDARDS (V. 1.0)

If a procedure within the package body is modified, the following information must appear after the existing procedure heading:

- Change Request or Discrepancy Report number
- Project Implementer and the date
- Description of change
- Changes to the procedure's algorithm
- Changes in the list of subprograms invoked by this procedure

The heading documentation must follow this format:

```
-- CHANGE REQUEST/DISCREPANCY REPORT NO:
-- PROJECT IMPLEMENTER:
-- DATE:
--
-- DESCRIPTION OF CHANGE: Briefly describe all changes made in the source code including how
-- it worked originally and how it works after the modification.
--
-- ALGORITHMS CHANGED: Describe the new algorithm if the old algorithm no longer applies to the
-- code.
--
-- SUBPROGRAMS MODIFIED: List the invoked subprograms that were changed.
```

5. In-line Documentation

When an error is repaired or a modification is made to the code, the original source code must be left in place, but preceded by the comment symbol, two consecutive hyphens (--). The original source code remains visible to Project Implementers, and serves as historical documentation. The Change Request or Discrepancy Report Number and an explanation of the effect of the change must be included. The in-line documentation of an error correction must have the following format:

```
-- ERROR REPAIR
-- DISCREPANCY REPORT NUMBER xxx
--
-- DESCRIPTION: Briefly describe the error and the change that was made.
-----
-- REMOVED: Oldcode
--      ...
--      ...
-----
-- ERROR REPAIR BEGINS
-----
--      Newcode      ...
-----
-- ERROR REPAIR ENDS
```

The in-line documentation of a modification must have the following format:

```
-- MODIFICATION
-- CHANGE REQUEST NUMBER xxx
--
-- DESCRIPTION: Briefly describe what is being modified.
-----
-- REMOVED: Oldcode
--      ...
--      ...
```

 -- MODIFICATION BEGINS

Newcode ...

 -- MODIFICATION ENDS

A. Revision History

Version 0.1	01/29/88	Janet Schnerre and Kathy Hoang
Version 0.2	02/28/88	Janet Schnerre and Kathy Hoang Made changes so that examples, acronyms, lists, and headings are consistent throughout document. Clarified differences between an error and a modification. Added Revision History material.
Version 0.3	03/17/88	Janet Schnerre and Kathy Hoang Corrected document format and spelling. Clarified reasons for in-line documentation.
Baselined	04/04/88	Bill Davis

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

QUALITY ASSURANCE PLAN

**QA Plan
04/04/88
Version 1.0**

**Written By
Sherrie A. Davis**

Table of Contents

1. Introduction.....	1
1.1. Acronyms.....	1
1.2. References.....	1
2. Quality Assurance Management.....	1
2.1. Organization.....	1
2.2. Responsibilities.....	1
3. Elements of the Quality Assurance Plan.....	2
3.1. Documentation Standards.....	2
3.2. Reviews.....	2
3.2.1. Code Reviews.....	2
3.2.2. Informal Reviews.....	2
3.2.3. Formal Reviews.....	2
4. Planned Reviews.....	3
4.1. Schedule.....	3
4.2. Documents.....	4
4.2.1. Requirements Document.....	4
4.2.2. Configuration Management Plan.....	4
4.2.3. Quality Assurance Plan.....	4
4.2.4. Coding Standards.....	4
4.2.5. Preliminary Design Document.....	5
4.2.6. Detailed Design Document.....	5
4.2.7. Test Plan.....	5
4.2.8. User Manual.....	5
5. Tool.....	5
6. Change Control Board.....	6
A. Revision History.....	6

1. Introduction

The purpose of this plan is to identify Quality Assurance (QA) management and to describe the standards and procedures implemented in the Quality Assurance Plan for the Documented Ada Style Checker (DASC) Project. The role of the Change Control Board in the QA process is discussed.

1.1. Acronyms

The following acronyms appear within the text of this plan:

CCB	Change Control Board
CDR	Critical Design Review
CM	Configuration Management
CR	Change Request
DASC	Documented Ada Style Checker
DR	Discrepancy Report
PDR	Preliminary Design Review
QA	Quality Assurance

1.2. References

The following documents are referenced in the text of this plan:

- Coding Standards
- Configuration Management Plan
- Detailed Design Document
- Documentation Standards and Guidelines
- Preliminary Design Document
- Requirements Document
- Test Plan
- User Manual

These documents are part of the DASC Project and can be obtained from the DASC Project team.

2. Quality Assurance Management

This section describes the organization and responsibilities of QA management.

2.1. Organization

The QA organization consists solely of the QA Manager.

2.2. Responsibilities

The QA Manager is responsible for the overall quality of the released product which includes the modified Style Checker software and project documentation. Primary responsibilities include the following:

- Preparing the QA Plan
- Calling and conducting formal reviews of documentation
- Monitoring software development using McCabe's cyclomatic complexity measure

3. Elements of the Quality Assurance Plan

This section describes documentation standards and methods to provide for compliance with established technical requirements.

3.1. Documentation Standards

Documentation Standards and Guidelines developed by the Document Specialist describe the conventions which pertain to all documentation associated with the DASC Project. Adherence to these guidelines will insure uniformity of all text documents. The Preliminary Design and Detailed Design Documents are exceptions to this policy. Because the tool *Excelsator* shall be used to produce these documents, they shall be in Excelsator report format.

After development, all documents will be considered in a formal review. After acceptance in the review process, the final form will be controlled by the Document Specialist to insure compliance with guidelines. Changes to documents shall be recorded in the Revision History section. The following information shall be listed for each version of the document:

- Version number
- Date the revision was made
- Author(s) of the revision
- Brief outline of the revision

3.2. Reviews

This section describes the informal and formal review methods used for quality assurance on the Documented Ada Style Checker (DASC) Project.

3.2.1. Code Reviews

Code reviews shall take the form of informal walkthroughs involving at least two people with one being a Project Implementer. All code shall pass a walkthrough before becoming part of the Style Checker source code.

3.2.2. Informal Reviews

Informal reviews shall take the form of discussions involving two or three members of the DASC Project team or Project Manager. Any member may request an informal review for such purposes as gaining another perspective, receiving expert advice, or reporting project progress. These informal reviews have no restrictions, and the QA Manager will have no responsibilities with respect to these reviews.

3.2.3. Formal Reviews

Formal reviews shall take the form of inspections involving four to six members of the DASC Project team. The QA Manager will schedule and conduct inspections. The QA Manager will insure a written record of the proceedings is kept, insure decisions made during the inspection are implemented, and determine if re-inspection is needed.

Inspections are the primary means of the QA Manager to check the DASC Project for compliance with established requirements. Therefore, the main goal of an inspection shall be to discover problems as soon as possible by utilizing the resources of the entire project team in an organized manner. Finding solutions to problems is not a function of inspections.

Specific roles will be assigned to participants. These roles and their responsibilities are as follows:

Leader	The QA Manager has this role. Responsibilities include scheduling, insuring all participants have prior access to products to be reviewed and adequate time to
--------	--

QUALITY ASSURANCE PLAN (V. 1.0)

consider them, moderating the discussion according to the rules which govern behavior, determining if inspection goals have been met, and reporting results.

- Recorder** This role is assigned on a review-by-review basis. Responsibilities include capturing inspection discussion in a concise, meaningful form, and producing a list of action items (Action List) for approval of the participants.
- Presenter** The person producing the product being reviewed has this role. Responsibilities include presenting an overview of the product and answering any questions. At re-inspections, the Presenter must address each item on the Action List indicating the change made or justifying the decision not to make the change.
- Reviewers** These roles are assigned on a review-by-review basis. Responsibilities include finding all errors in the product and preparing properly for the review.

Project Manager participation in inspections is limited to support by allotting sufficient time in the project schedule. Management absence at inspections helps establish a positive, non-threatening setting.

To facilitate a professional and productive atmosphere, inspections will be governed by specific rules of behavior. The rules emphasize that the product is reviewed, and not the producer of that product. Those rules which do not deal with the previously stated participant roles and responsibilities are as follows:

- Each Reviewer must make at least one critical comment and one compliment.
- No attack is to be made on the Presenter. The only comments allowed are those referencing the product.
- The Presenter will not respond to comments about the product except to ask for clarification of the comment.
- No solutions are to be presented. An inspection is held to uncover errors, not solve problems.
- The Leader will call upon Reviewers for comments. No one speaks unless called upon by the Leader.

4. Planned Reviews

This section describes the review schedule and documents which are subject to formal review.

4.1. Schedule

Four milestones have been established as follows:

February 9, 1988	Requirements Document Configuration Management Plan QA Plan Coding Standards
February 25, 1988	Preliminary Design Document
March 22, 1988	Detailed Design Document Test Plan
April 19, 1988	User Manual

Individual copies of all documents are to be made available to the inspection group at least one week prior to review dates.

4.2. Documents

Eight documents will be presented for formal review.

4.2.1. Requirements Document

The Requirements Document shall contain a detailed set of user requirements for the software and the external interfaces. It shall include the following:

- Functional requirements (what the system does)
- Physical requirements (how the system does it)
- Constraints (timing and performance needs, external interfaces with other systems, hardware, and operating system)

Since the intended audience is system users, no technical jargon shall be used. Each requirement shall be able to be validated. The document shall be complete with no implied requirements. The document must be signed by both the customer and Principal Architect. It can be changed only by renegotiation.

The Principal Architect shall prepare the Requirements Document and present it for formal review on February 9, 1988.

4.2.2. Configuration Management Plan

The Configuration Management (CM) Plan shall document the methods to be used for identifying the software product items, controlling and implementing changes, and recording and reporting change implementation status.

The CM Manager shall prepare the CM Plan and present it for formal review on February 9, 1988.

4.2.3. Quality Assurance Plan

The QA Plan shall contain the standards and procedures implemented to provide confidence that the DASC Project conforms to established technical requirements.

The QA Manager shall prepare the QA Plan and present it for formal review on February 9, 1988. In this review, the QA Manager will fill the role of Presenter instead of Leader.

4.2.4. Coding Standards

The Coding Standards shall contain the minimum style conventions expected of all source code produced by the DASC Project. Adherence to these standards will help insure the uniformity and maintainability of all source code.

The Project Implementers shall prepare the Coding Standards document and present it for formal review on February 9, 1988.

4.2.5. Preliminary Design Document

The Preliminary Design Document shall contain the architectural design specifications. Architectural design specifications shall establish a high level structural view of the system. They shall refine the conceptual view of the system, decompose high-level functions, and establish relationships and interconnections among packages, data types, and data operations. Packages shall be represented in a hierarchical diagram with textual descriptions of the main functions.

The Project Designer shall prepare the Preliminary Design Document and present it for formal review on February 25, 1988. The major goal of the Preliminary Design Review (PDR) shall be to insure the design represents an accurate baseline of the Style Checker before modification.

4.2.6. Detailed Design Document

The Detailed Design Document shall contain the internal design specifications. Internal design specifications shall describe the internal structure of each package and record design decisions. Each package will be exploded with graphs of each procedure. Structure charts for the procedures shall be included.

The Project Designer shall prepare the Detailed Design Document and present it for formal review on March 22, 1988. The goals of the Critical Design Review (CDR) will be basically the same as those for the PDR with the inclusion of additional design effort.

4.2.7. Test Plan

The Test Plan shall contain descriptions of functional, performance, and stress test cases to verify and validate the software as it develops, including tracing requirements through specifications, design, coding, and test. It shall specify the objectives of testing, the test completion criteria, and the methods used on particular modules. Modules which are modified shall be checked using the A-Test tool which includes Source Instrumentor, Automatic Path Analyzer, Performance Analyzer, Self-metric Analysis and Reporting Tool, and Path Analyzer.

The Project Tester shall prepare the Test Plan and present it for formal review on March 22, 1988.

4.2.8. User Manual

The User Manual shall contain descriptions of the required data and control inputs, input sequences, options, program limitations, and other items necessary for successful execution of the software. All error messages shall be identified and corrective actions described. A method of describing user-identified problems for software support shall be included.

The Document Specialist shall prepare the User Manual and present it for formal review on April 19, 1988. The goals of this review shall be to insure completeness, readability, and usability of the User Manual.

5. Tool

The McCabe cyclomatic metric is a software tool which computes complexity based on the control flow graph of the program. The tool builds the flow graph and computes the cyclomatic number for each subprogram body. Specifications do not have a complexity associated with them. In addition, it sums the number of nodes and edges in all program units and computes a cyclomatic number for the entire program. McCabe's cyclomatic complexity is defined as follows:

$$V(G) = e - n + 2p$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components.

Project Implementers shall use McCabe's cyclomatic complexity measure to monitor changes in the Style Checker source code. The complexity shall not increase by more than 2 without justification. The QA Manager shall monitor software development using McCabe's metric. If the established requirements have not been met, a Discrepancy Report (DR) shall be submitted to the Change Control Board.

6. Change Control Board

The Change Control Board (CCB) shall review all Change Requests (CR)s and Discrepancy Reports (DR)s. The Principal Architect, Project Administrator, CM Manager, and Document Specialist shall serve on the CCB. Each CCB meeting must have a quorum of three members. The Principal Architect shall call meetings as dictated by submission of CRs or DRs with a maximum of two meetings a week. At each meeting, pending CRs and DRs are approved or rejected with the Principal Architect having primary responsibility for decisions. The Project Administrator shall act for the Principal Architect in

QUALITY ASSURANCE PLAN (V. 1.0)

her absence. The Principal Architect shall publish a weekly summary of all actions taken. The CCB shall maintain separate files of Waived, Repaired, Pending, and new CRs and DRs. CRs and DRs may be submitted by any member of the DASC Project team to any CCB member or to the Project Manager.

A. Revision History

Version 0.1	02/02/88	Sherrie A. Davis
Revision 0.2	03/15/88	Sherrie A. Davis
History.	Added sections 1.1. Acronyms, 1.2. References, 5.0. Tool, and A. Revision Added inspection rules in section 3.2.3. Added new information concerning the CCB in section 6.0.	
Baselined	04/04/88	Bill Davis

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

TEST PLAN

**Test Plan
04/29/88
Version 1.0**

**Written By
Scott Meyer**

Table of Contents

1. Introduction.....	1
1.1. Scope.....	1
1.2. Acronyms.....	1
1.3. References.....	1
2. Test Items.....	1
3. Requirements To Be Tested.....	2
4. Requirements Not To Be Tested.....	2
5. Testing Approach.....	3
5.1. System Testing.....	3
5.2. Regression Testing.....	3
6. Responsibilities.....	3
A. Revision History.....	3
Appendix. Test Cases..	5

1. Introduction

The purpose of this Test Plan is to define and describe the activities and procedures to be implemented in the formal testing of the Documented Ada Style Checker (DASC) Project. This plan is structured according to the DASC Documentation Standards and Guidelines.

1.1. Scope

This plan covers system testing of DASC. Also covered in this plan is the performing of regression testing on DASC.

1.2. Acronyms

The following acronyms are referred to in this plan:

DASC	Documented Ada Style Checker
DEC VAX	Digital Equipment Corporation Virtual Address eXtended
CCB	Change Control Board
CM	Configuration Management
DR	Discrepancy Report
IPFW	Indiana University Purdue University at Fort Wayne

1.3. References

The following documents are referred to in this plan:

- Documentation Standards and Guidelines
- Requirements Document

The above documents are part of the DASC Project and may be obtained from the DASC Project group.

2. Test Items

The test cases (programs) reside in the Digital Equipment Corporation's Virtual Address eXtended (DEC VAX) directory [C474A7.TEST]. The names of the files containing test cases and the corresponding flaw and style reports for version 1.0 of the Style Checker are as follows:

TEST1.ADA;1, TEST1.FLW;1, TEST1.STY;1
TEST2.ADA;1, TEST2.FLW;1, TEST2.STY;1
TEST3.ADA;1, TEST3.FLW;1, TEST3.STY;1
TEST4.ADA;1, TEST4.FLW;1, TEST4.STY;1

If the Style Checker is updated or modified, the new versions of the flaws and style report files will be renamed to reflect the new version of the Style Checker. The renaming will be as follows:

TEST1_1.FLW, TEST1_1.STY
TEST2_1.FLW, TEST2_1.STY

These are modified test results for version 1.1 of the Style Checker.

TEST1_2.FLW, TEST1_2.STY
TEST2_2.FLW, TEST2_2.STY

These are modified test results for version 1.2 of the Style Checker.

3. Requirements To Be Tested

TEST PLAN (V. 1.0)

The following requirements validation matrix lists the style reporting requirements specified in the Requirements Document (Section 3.3) that will be tested and the corresponding test case that covers each requirement. The test cases can be found in the appendix of this document.

Requirement	Test Case
3.3.1.1 Invalid Object Case	1
3.3.1.2 Invalid Keyword Case	1
3.3.1.3 Name Segment Size	1
3.3.1.4 Average Name Size	1
3.3.2.1 Occurrences of Multiple Statements Per Line	2
3.3.2.2 Inconsistent Indentation	2
3.3.2.3 Missing Blank Lines	2
3.3.2.4 Loops Without Names	2
3.3.3.1 Percent of Literal Usage	3
3.3.3.2 Percent of Universal Type Usage	3
3.3.3.3 Attribute Use	3
3.3.3.4 Ada Specific Features Used	3
3.3.4.1 Number of Subprogram Parameters	4
3.3.4.2 Subprogram Size	1
3.3.4.3 Number of Loop Exits	4
3.3.4.4 Nesting Levels	4
3.3.5.1 Number of Comments	3
3.3.5.2 Average Comment Size	3
3.3.6.1 Line Length Violations	3
3.3.6.2 Address Clauses Used	4
3.3.6.3 Pragmas Used	3
3.3.6.4 Representation Specification Used	3
3.3.7.1 Keywords Used	2
3.3.7.2 Keywords Allowed	2

4. Requirements Not To Be Tested

The user interface requirements specified in the Requirements Document (Section 3.1) shall be tested by the entire DASC group. The user interface or shell is a project developed environment for running the Style Checker. Each DASC team member will test the shell and make recommendations in writing to the Project Implementers. Each recommendation shall be answered until all members have been satisfied.

The style definition requirements specified in the Requirements Document Section 3.2) shall not be tested by the Project Tester. The Project Tester will assume that the style convention used at Indiana University - Purdue University at Fort Wayne (IPFW) has been previously defined in the DASC tool. Only the IPFW style convention will be tested.

5. Testing Approach

Two main testing approaches will be taken by the Project Tester. They are system testing and regression testing.

5.1. System Testing

TEST PLAN (V. 1.0)

Black-box (functional, requirements based) testing is a method in which test cases are derived from an analysis of the system's functional specifications. System testing involves the black-box testing of the entire DASC system. System testing shall be done by executing the Style Checker on the test cases, TestN.adb where N is 1,2,3,4, residing in the DEC VAX directory [C474A7.TEST]. The test cases will verify and validate the style reporting requirements specified in the Requirements Document by showing valid and invalid equivalence classes for each requirement.

5.2. Regression Testing

Regression testing is performed after a modification or correction on the DASC system has been made. Individual unit or module testing shall be done by the Project Implementers after a modification or correction.

Upon unit testing, the Project Implementers will notify the Configuration Management (CM) Manager which module(s) have been changed.

After DASC has been reconfigured, the CM Manager will notify the Project Tester that a new version of DASC is ready to be tested and will specify which modification or correction has been made. The Project Tester will then determine which test case(s) specifically cover the change(s) that have been made and execute the Style Checker on these test case(s) to verify that the change has been performed correctly. If the change is found not to perform correctly, then the Project Tester will submit a Discrepancy Report (DR) to the Change Control Board (CCB). If the changed module(s) performs correctly, the remaining test cases will be performed and the flaw and style reports produced will then be compared to the original results to verify that no differences exist. If a difference other than the intended change is found, a DR will be submitted to the CCB by the Project Tester. If the system regression test produces correct results, the Project Tester will notify the CM Manager.

6. Responsibilities

The Project Tester will be responsible for managing and executing the testing activities. Primary responsibilities include the following:

- Preparing the Test Plan
- Developing the test cases for DASC
- Preparing baseline system test results
- Testing modified code using isolated test cases
- Doing regression testing on the system after each modification or correction

A. Revision History

Version 0.1	03/31/88	Scott Meyer
Revision 0.2	04/24/88	Scott Meyer Added test cases to the Appendix. Completed the requirements validation matrix. Added names of files containing test cases to section 2.0. Added definition of a shell to section 4.0. Added definition of black-box testing to section 5.1. Made wording changes and corrected spelling mistakes throughout document. Added responsibilities to section 6.0.
Revision 0.3	04/29/88	Scott Meyer Made additional wording changes and corrected spelling errors.

Appendix. Test Cases

[Editor's Note: Seven test cases were delivered with this document. The correspondences between these seven and the four test cases mentioned in sections 2 and 3 are not known.]

```
--
--  T E S T   1
--
-- This module is used to test the Coding Standards Checker for ADA - The
-- tests that are to be applied to the module are Test A1 indices 1 thru 7
-- and 14. Various additions have been added to a module originally named
-- Menu.adb authored by Jerry Baskette. Changes were made January 30, 1985
-- by James Rea.
--
-- Additions and changes have been commented throughout the module. These
-- comments relate the tests to the code.

with CURRENT_EXCEPTION;
separate (HELP_UTILITY)

-----
-- The reserved word 'procedure' is in upper case
-- This should be flagged as improper case - Test A1 index 1
-----

PROCEDURE PRINT_TOPIC_MENU (NODE: in HELP_LINK) is

  LINE_COUNT:          TEXT_RANGE := 0;  -- number of lines to be output
  TOTAL_NUMBER_OF_TOPICS:  NATURAL := 0;

  -- The name NUM_TOPICS_IN_COLUMN_ONE in the wrong case

  -----
  -- This should be flagged as improper case - Test A1 index 2
  -----

  num_TOPICS_IN_COLUMN_ONE: POSITIVE := 1;
  TOPICS_IN_COLUMN_TWO:      BOOLEAN := FALSE;
  OUTPUT_LINE:               HELP_INFO_TEXT_LINE;
  OUTPUT_BUFFER:             SAVED_TEXT; -- buffer of OUTPUT_LINES to be output
  RIGHT_COLUMN_START:        INTEGER := (MAX_LINE_LENGTH/2) + 2;

  -----
  -- The name EVEN should flag the short word (length 4)
  -----

  EVEN:                     BOOLEAN := FALSE;

  -----
  -- The name 'EVENS' should not flag the short name defn. Test A1 index 6
  -----

  EVENS:                     POSITIVE := 1;

  -----
  --The names 'I,J,K' should not be flagged. Test A1 index 7
  -----
```


TEST PLAN (V. 1.0)

```
I:          POSITIVE;
J:          POSITIVE;
K:          POSITIVE;
TESTFLAG:   BOOLEAN;
TEST_FLAG:  BOOLEAN;
```

```
-----
--The name 'TSTFLG' should be flagged as an abbreviation
--Test A1 index 14
-----
```

```
TSTFLG:      BOOLEAN;
TEST_FLAG:   BOOLEAN;
TESTFLAGS:   BOOLEAN; --should be flagged for indentation
  TEST_FLAGS: BOOLEAN;  --should be flagged for indentation
CURRENT_NODE:  HELP_LINK := null;--flag for indentation
COL_ONE_NODE:  HELP_LINK := null;--flag for indentation
COL_TWO_NODE:  HELP_LINK := null;--flag for indentation
```

begin

```
-----
--Indentation should be checked per Test A2
-----
```

```
CURRENT_NODE := NODE.SUBTOPICS;
-- count the number of subtopics for this node
```

```
while CURRENT_NODE /= null loop
  TOTAL_NUMBER_OF_TOPICS := TOTAL_NUMBER_OF_TOPICS + 1;
  CURRENT_NODE:= CURRENT_NODE.NEXT_TOPIC;
end loop;
```

```
-- If there is more than one topic, then split the topics into two columns
-- Column one will have the first half and column two the second half.
-- If there are an odd number of topics, column one will have the odd number
```

```
-----
--The following 'if statements should be flagged as two statements
--on one line.  Test A2 index 1
-----
```

```
if TOTAL_NUMBER_OF_TOPICS /= 0 then if TOTAL_NUMBER_OF_TOPICS >= 2 then
  TOPICS_IN_COLUMN_TWO := TRUE;
  NUM_TOPICS_IN_COLUMN_ONE := TOTAL_NUMBER_OF_TOPICS / 2;
```

```
-- More than one topic, split the number
--See if odd number. If so, increment the topic count so odd goes in 1st col.
```

```
  if TOTAL_NUMBER_OF_TOPICS /= (TOTAL_NUMBER_OF_TOPICS/2) * 2 then
    NUM_TOPICS_IN_COLUMN_ONE := NUM_TOPICS_IN_COLUMN_ONE + 1;
```

```
  else
```

```
-----
--A second test of multiple statements on one line.
-----
```

TEST PLAN (V. 1.0)

```
    EVEN := TRUE; end if; end if; --three statements on this line!

    -- set the beginning node for each column

COL_ONE_NODE := NODE.SUBTOPICS;
CURRENT_NODE := NODE.SUBTOPICS;

-----
--The following 'for' statement should flag improper indentation
--Test A2 index 3
-----

for I in 1..NUM_TOPICS_IN_COLUMN_ONE loop
COL_TWO_NODE := CURRENT_NODE.NEXT_TOPIC;
CURRENT_NODE := CURRENT_NODE.NEXT_TOPIC;
end loop;
-----
--The following 'while' and 'if' statements should be flagged for
--improper indentation. Test A2 index 3
--Lack of blank line following the above 'for' loop should be flagged.
--This lack of blank line preceeding the 'if' below should also
--should also be flagged. Text A2 indices 8 & 9 respectively.
-----
if TOPICS_IN_COLUMN_TWO then
while COL_TWO_NODE /= null loop
OUTPUT_LINE := BLANK_LINE; -- blank the line buffer

-- Put first topic in left half of output line

OUTPUT_LINE(1..COL_ONE_NODE.NAME_LENGTH) :=
    COL_ONE_NODE.NAME(1..COL_ONE_NODE.NAME_LENGTH);

-- Put second topic in second half of output line

    OUTPUT_LINE(RIGHT_COLUMN_START..RIGHT_COLUMN_START +
COL_TWO_NODE.NAME_LENGTH - 1) :=
        COL_TWO_NODE.NAME(1..COL_TWO_NODE.NAME_LENGTH);

-- Put the line in the output buffer. LINE_COUNT is incremented automatically

-----
--This is a block of comments to check the proper indentation of
--the block of comments. This is done in accordance with Test A2
--index 6. This should be flagged.
-----

    HELP_INFO_SUPPORT.APPEND_TO_DISPLAY(OUTPUT_LINE,OUTPUT_BUFFER,LINE_COUNT);
    COL_ONE_NODE := COL_ONE_NODE.NEXT_TOPIC;
    COL_TWO_NODE := COL_TWO_NODE.NEXT_TOPIC;
end loop;
end if;

if not EVEN then

-- Put the odd topic in the output buffer
```

TEST PLAN (V. 1.0)

```
OUTPUT_LINE := BLANK_LINE;
OUTPUT_LINE(1..COL_ONE_NODE.NAME_LENGTH) :=
  COL_ONE_NODE.NAME(1..COL_ONE_NODE.NAME_LENGTH);
HELP_INFO_SUPPORT.APPEND_TO_DISPLAY(OUTPUT_LINE,OUTPUT_BUFFER,LINE_COUNT);
end if;
```

```
-- Now print the output buffer
```

```
HELP_INFO_SUPPORT.PRINT_TEXT(OUTPUT_BUFFER,LINE_COUNT);
end if;
```

```
-----
--Lack of blank lines separating the major block (exception) should
--be flagged.  Test A2 index 7.
-----
```

```
exception
  when others => text_io.put_line("Print Menu " & CURRENT_EXCEPTION.NAME);
  raise;
end PRINT_TOPIC_MENU;
```

```
-----
--
--  T E S T 1 A
--
-----
```

```
-- Test for Short names
-----
```

```
procedure XYZ ( Z : in INTEGER; Y : OUT FLOAT ) is
begin
  return;
end XYZ;
```

```
-----
--
--  T E S T 1 B
--
-----
```

```
-- Test for UNDERSCORES -- the result should flag
-- this program as having not enough underscores,
-- i.e. the name segment size too small!
-----
```

```
procedure XYZANDPDQ_FOR_LONG_NAME (
  ZYZENZUZEN_ZULULONG : in INTEGER;
  YYILLERBYT_YTHISIS_ALSOTOOLONG : OUT FLOAT ) is
begin
  return;
end XYZANDPDQ_FOR_LONG_NAME;
```

```
-----
--
--  T E S T 2
--
-----
```

```
package body INITIALIZE is
begin
  -----
```

TEST PLAN (V. 1.0)

--The following assignments should raise flags for Test A4
 --The assignments that show 1 and 0 should not be flagged for Test A4

```

OPENED := FALSE;
CHECK := 5;
CHECKONE := 1;      --should not be flagged
CHECKONE := 0;      --should not be flagged
CHECK := 150;
CHECK := 32767;
CHECKTWO := "FIVE";
CHECKTWO := "FOUR";

WHILE not OPENED loop
  FILE_IO.OPEN_INPUT_FILE(INPUT_FILES(FILE_LEVEL).FILE, OPENED, DUMMY);
end loop;
if TEXT_IO.IS_OPEN(INPUT_FILES(FILE_LEVEL).FILE) then
  OPENED := FALSE;
  while not OPENED loop
    FILE_IO.OPEN_OUTPUT_FILE(OUTPUT_FILE,
      TEXT_IO.NAME(INPUT_FILES(FILE_LEVEL).FILE), OPENED);
  end loop;
  HOST_LCD_IF.GET_TIME(TODAY, TIME_ERROR);
  if TIME_ERROR = HOST_LCD_IF.NOT_AN_ERROR then
    TIME_STAMP := "          ";
    TEXT_IO.INTEGER_IO.PUT(TIME_STAMP(11..12), TODAY.MONTH);
    TIME_STAMP(13) := '/';
    TEXT_IO.INTEGER_IO.PUT(TIME_STAMP(14..15), TODAY.DAY);
    TIME_STAMP(16) := '/';
    TEXT_IO.INTEGER_IO.PUT(TIME_STAMP(17..20), TODAY.YEAR);
    SECONDS_SINCE_MIDNIGHT := TODAY.TICKS_SINCE_MIDNIGHT /
      TODAY.TICKS_PER_SECOND;
    HOURS := SECONDS_SINCE_MIDNIGHT / SECONDS_PER_HOUR;
    MINUTES := ((SECONDS_SINCE_MIDNIGHT) - (SECONDS_PER_HOUR * HOURS)) /
      SECONDS_PER_MINUTE;
    SECONDS := ((SECONDS_SINCE_MIDNIGHT) - (SECONDS_PER_HOUR * HOURS)) -
      (SECONDS_PER_MINUTE * MINUTES);
    TEXT_IO.LONG_INTEGER_IO.PUT(TIME_STAMP(1..2), HOURS);
    TIME_STAMP(3) := ':';
    TEXT_IO.LONG_INTEGER_IO.PUT(TIME_STAMP(4..5), MINUTES);
    TIME_STAMP(6) := ':';
    TEXT_IO.LONG_INTEGER_IO.PUT(TIME_STAMP(7..8), SECONDS);
  else
    TIME_STAMP := "          ";
  end if;
end if;
end INITIALIZE;

```

```

--
--  T E S T 3A
--

```

 -- This is a modification of Test 3 to include ALL Ada keywords!

```

with TEXT_IO;
procedure INSERT      (LENGTH : in ID_RANGE;
                      STRG : in ID_STRING;

```

TEST PLAN (V. 1.0)

```

                                T : in out TREE;
                                REFERENCE_LINE_NUMBER : in LINE_NUM_RANGE) is
FOLLOW_CHAIN : REFPTR;          -- used to follow
                                -- chain of references
TESTER       : BOOLEAN := TRUE;
```

```
-----
--The following package declarations and package bodies have been
--added to this module for testing purposes.
-----
```

package CHECKER is

```

    task INTERRUPT is
        entry DONE;
        for DONE use at 16#40#;

        function HELP return BOOLEAN is separate;

        subtype XDATA is new REALS digits 4;
        type YDATA is REALS digits 4 range 0.0 .. 100.0;
        LIMIT : constant INTEGER := 3;
```

```

    type INFO is
        record
            VERTICAL : YDATA;
            HORIZONTAL : XDATA
        end record;
    limited private
```

```

    type NEW_PTR is access INFO;
```

```

    OOPS : exception;
    generic
    package COUNTER is
    begin
        return;
    end COUNTER;
begin
```

```

LOOP_NAME:
    loop
        <<INTERRUPT_NAME>>
        accept INTERRUPT do
            delay 60.0;
            GET_IT(INFO.all);
        end INTERRUPT;
```

```

    declare
        EXTRA : exception renames OOPS;
        FRACT : FRACTION delta 0.0001;
        SAVER : array(0..1) of BOOLEAN;
```

```

    begin
        for I in reverse 1..31 loop
            if abs(INFO.HORIZONTAL) mod 30 = 0 and then
                not HELP
```

TEST PLAN (V. 1.0)

```
        or else INFO.VERTICAL rem 30 then
            abort OTHER_TASK;
        elsif HELP xor TRUE then
            goto INTERRUPT_NAME;
        end if;
    end loop;
end;

case HELP is
    when others =>
        exit;
end case;

select
    when HELP =>
        exit;
    when others =>
        raise OOPS;
end select;
end loop LOOP_NAME;
terminate;
end INTERRUPT;

-----
-- The following should be flagged on the inner packages
-- as being nested too deeply!
-----

package CHECKER_ONE is
    package CHECKER_TWO is
        package CHECKER_THREE is
            procedure TRY_AGAIN;
        end CHECKER_THREE;
    end CHECKER_TWO;
end CHECKER_ONE;

end CHECKER;

package body CHECKER is

    package body CHECKER_ONE is
        package body CHECKER_TWO is
            package body CHECKER_THREE is
                procedure TRY_AGAIN is
                    begin
                        null;
                    end TRY_AGAIN;
                end CHECKER_THREE;
            end CHECKER_TWO;
        end CHECKER_ONE;

    end CHECKER;

-----
--The following procedure calls have been added to check for nesting
--levels.
```

TEST PLAN (V. 1.0)

```
-----
procedure STUFFER is
  procedure STUFFERONE is
    procedure STUFFERTWO is
      procedure STUFFERTHREE is
        procedure STUFFERFOUR is
          begin
            TEXT_IO.PUT_LINE("In stufferfour!");
          end STUFFERFOUR;

        begin
          TEXT_IO.PUT_LINE("In stufferthree!");
        end STUFFERTHREE;

        begin
          TEXT_IO.PUT_LINE("In stuffertwo!");
        end STUFFERTWO;

        begin
          TEXT_IO.PUT_LINE("In stufferone!");
        end STUFFERONE;

        begin
          TEXT_IO.PUT_LINE("In stuffer!");
        end STUFFER;
      end STUFFERONE;
    end STUFFERTWO;
  end STUFFERTHREE;
end STUFFERTWO;
end STUFFERONE;
end STUFFER;

begin
  if T = null then          -- add this identifier and reference here
    T := new NODE'(STRG,LENGTH,null,null,null);
    T.REFERENCES := new REFS'(REFERENCE_LINE_NUMBER,null);
  else
    if T.STRG = STRG then   -- new reference to old identifier
      FOLLOW_CHAIN := T.REFERENCES;
      while FOLLOW_CHAIN.NEXT /= null loop
        FOLLOW_CHAIN := FOLLOW_CHAIN.NEXT;
      end loop;
    end if;
  end if;

  -----
  --The following loops are added to test nesting levels of loops.
  -- The outer loop should be flagged as nested too deep.
  -- The inner loop should be flagged as needing a loop name.
  -----

  while TESTER loop
    while TESTER loop
      while TESTER loop
        for INDEX in 1..100 loop
          while TESTER loop
            while TESTER loop
              -- should be flagged - nesting too deep
              while TESTER loop
                null;
              end loop;
            end loop;
          end loop;
        end loop;
      end loop;
    end loop;
  end loop;
end;
```

TEST PLAN (V. 1.0)

```

        end loop;
    end loop;
    TESTER := FALSE;
    end loop;
    end loop;
    end loop;
    TESTER := TRUE;
    if FOLLOW_CHAIN.REFNUM /= REFERENCE_LINE_NUMBER then
        FOLLOW_CHAIN.NEXT := new REFS'(REFERENCE_LINE_NUMBER,null);
    end if;
else
    if T.STRG > STRG then
        INSERT(LENGTH,STRG,T.LEFT,REFERENCE_LINE_NUMBER);
    else
        INSERT(LENGTH,STRG,T.RIGHT,REFERENCE_LINE_NUMBER);
    end if;
end if;

-----
--The following if's are inserted for testing purposes.
-----

if TESTER then
    if TESTER then
        if TESTER then
            if TESTER then
                null;
            end if;
            null;
        end if;
        null;
    end if;
    null;
end if;
end if;
end INSERT;
pragma MAIN;
```

```

--
-- T E S T  4
--
-----PROLOGUE-----
--
-- Unit name      : next_character
-- Author         : Richard D. Powers
-- Date created   : 7-29-83
-- Last update    :
--
--
-----
--
-- Abstract       : This procedure returns the next character from the
-----: input stream. If the character is a lower case
-----: letter then it is converted to upper case.
--
-----
--
```


TEST PLAN (V. 1.0)

```

-- Inputs      : INPUT_FILES - global file record
-- Outputs     : CH - next character from input stream
-- Procedures called : GET_LINE, NEXT_CHARACTER
-- Exceptions  : If end of file is encountered then the return
--               character is not changed, but current line will
--               have it's length set to -1.
--
-----
--
-- Mnemonic      :
-- Name          :
-- Release date  :
-----
-- Revision history -----
--
-- DATE          AUTHOR          HISTORY
--
--
--
-----END-PROLOGUE-----

with TEXT_IO;
procedure NEXT_CHARACTER (CH : out CHARACTER) is
--pragma INCLUDE("GETLINE.ada.");
-----PROLOGUE-----
--
-- Unit name     : get_line
-- Author        : Richard D. Powers
-- Date created  : 7-29-83
-- Last update   : 10-07-83
--
--
-- Abstract      : This procedure gets the next line from the input
--               : stream.
--
--
--
-----
--
-- Inputs       : INPUT_FILE - global file record
-- Outputs      : LINE - next line from input stream
-- Procedures called : PUT_HEADER, TEXT_IO.END_OF_FILE,
--                   TEXT_IO.END_OF_LINE, TEXT_IO.GET,
--                   TEXT_IO.INTEGER_IO.PUT, TEXT_IO.NEW_LINE
--                   TEXT_IO.PUT, TEXT_IO.SKIP_LINE
-- Exceptions    : If end of file is encountered then line length is
--               : set to -1.
--
-----
--
-- Mnemonic      :
-- Name          :
-- Release date  :
-----
-- Revision history -----
--
-- DATE          AUTHOR          HISTORY
-- 10-07-83      R. Powers        Don't read a character at a time
--
--
-----END-PROLOGUE-----

```

TEST PLAN (V. 1.0)

```

procedure GET_LINE(LINE : out LINE_RECORD) is
  NEWLINE : STRING(1..MAX_LINE_LENGTH);      -- parameter for text_io routines

  LOOP_FLAG, LOOP_FLAG_ONE : BOOLEAN := true;

begin

  -----
  --A loop added here for testing purposes.
  -----

  for INDEX in 1..1 loop --long loop (no name) should be flagged.
    LOOP_TEST: --long loop with a name (no flag)
    for INDEX in 1..1 loop
      if TEXT_IO.END_OF_FILE(INPUT_FILES(FILE_LEVEL).FILE) then
        LINE.LENGTH := -1;
        LINE.COLUMN := 0;
      else
        if LINE_NUMBER mod 40 = 0 then
          TEXT_IO.NEW_LINE;
        end if;
        if LINE_NUMBER mod 58 = 0 then
          PUT_HEADER;
        end if;
        TEXT_IO.PUT('.');
        LINE_NUMBER := LINE_NUMBER + 1;
        TEXT_IO.INTEGER_IO.PUT(OUTPUT_FILE, LINE_NUMBER, 5);
        if LOOP_FLAG then
          exit;
        end if;
        if LOOP_FLAG_ONE then
          exit LOOP_TEST;
        end if;
        while LOOP_FLAG loop --Multiple exits should be flagged.
          LOOP_FLAG := false;
          if LOOP_FLAG then
            exit LOOP_TEST;
          end if;
          exit;
        end loop;
        exit LOOP_TEST;
        TEXT_IO.PUT(OUTPUT_FILE, ' ');
        TEXT_IO.GET_LINE(INPUT_FILES(FILE_LEVEL).FILE, NEWLINE, LINE.LENGTH);
        if LINE.LENGTH > 75 then -- only write first 75 characters
          TEXT_IO.PUT(OUTPUT_FILE, NEWLINE(1..74));
        else
          TEXT_IO.PUT(OUTPUT_FILE, NEWLINE(1..LINE.LENGTH));
        end if;
        for I in 1..LINE.LENGTH loop
          LINE.LINE(I) := NEWLINE(I); --No exits in loop (no flags)
        end loop;
        LINE.COLUMN := 0;
        TEXT_IO.NEW_LINE(OUTPUT_FILE);
      end if;
    end loop LOOP_TEST;
  end loop;
end GET_LINE;

```

TEST PLAN (V. 1.0)

```

begin
  if INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN <
    INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH then
    INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN :=
      INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN + 1;
    CH := INPUT_FILES(FILE_LEVEL).CURRENT_LINE.
      LINE(INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN);
  elsif INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH > -1 then
    GET_LINE(INPUT_FILES(FILE_LEVEL).CURRENT_LINE);
    NEXT_CHARACTER(CH);
  end if;
  if (CH >= 'a') and (CH <= 'z') then
    CH := CHARACTER'VAL(CHARACTER'POS(CH) - CHARACTER'POS('a') +
      CHARACTER'POS('A'));
  end if;
end NEXT_CHARACTER;

```

```

--
-- T E S T 5
--
-----
--This procedure has been altered in order to provide tests for the
--Style Checker program. Line length must be set by the parameters
--of the Checker with a maximum length of 80 characters
-----
--PROLOGUE-----
--
-- Unit name      : next_identifier
-- Author         : Richard D. Powers
-- Date created   : 7-29-83
-- Last update    :
--
--
--
-----
-- Abstract      : This procedure returns the next identifier in the
--                : input stream.
--
--
-----
--
-- Inputs        : INPUT_FILES - global input file record
-- Outputs       : IDENTIFIER - next identifier from input stream
--                LENGTH - length of next identifier
--                REFERENCE_LINE_NUMBER - line number of this id
-- Procedures called : NEXT_CHARACTER
-- Exceptions     : if no identifier is found (i.e. end of file) then
--                LENGTH is set to zero.
--
--
-----
--
-- Mnemonic      :
-- Name          :
-- Release date   :
-----
-- Revision history -----
--
-- DATE          AUTHOR          HISTORY
--
--
--

```

TEST PLAN (V. 1.0)

-----END-PROLOGUE-----

```
with CURRENT_EXCEPTION;
with TEXT_IO;
procedure NEXT_IDENTIFIER(IDENTIFIER : out LINE_STRING;
                           LENGTH : out LINE_INDEX_RANGE;
                           REFERENCE_LINE_NUMBER : out LINE_NUM_RANGE) is
```

-----The following is added to test the flagging of representation specs.-----

```
type MEDIUM is range 0..100;
BYTE : constant := 8;
for MEDIUM'SIZE use 2*BYTE;
```

```
type COLOR is (RED,BLUE,YELLOW);
for COLOR use (RED => 5,BLUE => 10,YELLOW => 15);
```

```
CH : CHARACTER; -- get a character at a time
INDEX : INTEGER range 0..MAX_LINE_LENGTH; -- index into identifier
```

-----The following is a test for recognition of Pragmas-----

```
pragma LIST(OFF);
pragma LIST(ON);
pragma OPTIMIZE(SPACE);
pragma PAGE;
begin
  for I in 1..ID_LENGTH loop
    IDENTIFIER(I) := ' ';
  end loop;
  CH := ' '; -- force entry into first loop
  WHILE not ((CH >= 'A') and (CH <= 'Z'))
    and (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH > -1) loop
    NEXT_CHARACTER(CH);
    if CH = '-' then -- look for comment indicator
      INDEX := INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN;
      -- remember which column we are at
    NEXT_CHARACTER(CH);
    if (CH = '-') and
      (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN = INDEX + 1)
    then -- found a comment
      while INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN <
        INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH loop
        NEXT_CHARACTER(CH);
      end loop;
      CH := ' '; -- force us to remain in loop
    end if;
  end if;
```

-----This is a long line to allow for the maximum parameter of the StyleChecker-----

TEST PLAN (V. 1.0)

```

if (CH >= '0' and CH <= '9') then -- skip over literals
  NEXT_CHARACTER(CH);
  while ((CH >= '0' and CH <= '9') or (CH = 'E') OR (CH = '_')) and
    (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN <
      INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH)
    loop
      NEXT_CHARACTER(CH);
      if (CH = '#') then -- skip over based literals
        NEXT_CHARACTER(CH);
        WHILE (CH /= '#') and
          (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN <
            INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH)
          loop
            NEXT_CHARACTER(CH);
          end loop;
        NEXT_CHARACTER(CH);
      end if;
    end loop;
  end if;
  if (CH = '"') then -- skip over string constants
    NEXT_CHARACTER(CH);
    while (CH /= '"') and
      (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN <
        INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH)
      loop
        NEXT_CHARACTER(CH);
      end loop;
  end if;
  if ((CH = '"') and
    (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN + 2 <=
      INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH)) and then
    INPUT_FILES(FILE_LEVEL).CURRENT_LINE.
      LINE(INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN + 2) = '"'
    then
      NEXT_CHARACTER(CH); -- skip character constant
      NEXT_CHARACTER(CH); -- skip end of constant delimiter
    end if;
  end loop;
  if INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH = -1 then
    LENGTH := 0; -- end of file encountered
  else
    if INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN =
      INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH
    then -- last character on line
      IDENTIFIER(1) := CH;
      LENGTH := 1;
      REFERENCE_LINE_NUMBER := LINE_NUMBER;
    else
      REFERENCE_LINE_NUMBER := LINE_NUMBER;
      INDEX := 0;
      while (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN <
        INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH)
        and (((CH >= 'A') and (CH <= 'Z')) or (CH = '_') or
          ((CH >= '0') and (CH <= '9')))) loop
        INDEX := INDEX + 1;
        IDENTIFIER(INDEX) := CH;
        NEXT_CHARACTER(CH);
      end loop;
    end if;
  end if;
end if;

```

TEST PLAN (V. 1.0)

```

if (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN =
    INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH)
    and (((CH >= 'A') and (CH <= 'Z')) OR
        ((CH >= '0') and (CH <= '9')) or (CH = '_')) then
    INDEX := INDEX + 1;
    IDENTIFIER(INDEX) := CH;
end if;
end loop;
if INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN /=
    INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH
then
    INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN :=
        INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN - 1;
end if;
if INDEX > ID_LENGTH then
    LENGTH := ID_LENGTH;
else
    LENGTH := INDEX;
end if;
end if;
end if;

-----
--A call to a proscribed package
-----
TEXT_IO.PUT(CURRENT_EXCEPTION.NAME);
-----

end NEXT_IDENTIFIER;

-----

pragma MAIN;
-----

-----

--
-- T E S T   6
--
-----PROLOGUE-----
--
-- Unit name      : push_file
-- Author         : Richard D. Powers
-- Date created   : 7-29-83
-- Last update    : 1-31-84
--
--
--
-- Abstract       : This procedure handles the pragma include, opening
--                  : the include file and pointing current input to it.
--
--
--
-- Inputs         : INPUT_FILES - global file record
-- Outputs        : INPUT_FILES - global file record
-- Procedures called : NEXT_CHARACTER, FILE_IO.OPEN_INPUT_FILE,
--                  : TEXT_IO.NEW_LINE, TEXT_IO.PUT
-- Exceptions      :
--
--
-----

```

TEST PLAN (V. 1.0)

```
-- Mnemonic      :
-- Name          :
-- Release date   :
----- Revision history -----
--
-- DATE          AUTHOR          HISTORY
-- 1-31-84       R. Powers       Open routines moved to own package
--
--
-----END-PROLOGUE-----

procedure PUSH_FILE is
  CH : CHARACTER;          -- read a character at a time
  FILE_NAME : STRING(1..MAX_LINE_LENGTH); -- name of new input file
  FILE_NAME_LENGTH : INTEGER range 0..MAX_LINE_LENGTH := 0;
                                -- length of name
  OPENED : BOOLEAN;        -- parameter to open_input_file
begin
  for I in 1..FILE_NAME'LAST loop
    FILE_NAME(I) := ' ';
  end loop;
  NEXT_CHARACTER(CH);
  WHILE CH = ' ' loop
    NEXT_CHARACTER(CH);
  end loop;
  if CH /= '(' then
    TEXT_IO.PUT("ERROR IN PARSING PRAGMA INCLUDE - MISSING '('");
    TEXT_IO.NEW_LINE;
  else
    NEXT_CHARACTER(CH);
    while CH = ' ' loop
      NEXT_CHARACTER(CH);
    end loop;
    if (CH /= '"') then
      TEXT_IO.PUT("ERROR IN PARSING PRAGMA INCLUDE - MISSING '\"'");
      TEXT_IO.NEW_LINE;
    else
      NEXT_CHARACTER(CH);
      while (CH /= '"') and
        (INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN <
         INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH)
      loop
        FILE_NAME_LENGTH := FILE_NAME_LENGTH + 1;
        FILE_NAME(FILE_NAME_LENGTH) := CH;
        NEXT_CHARACTER(CH);
      end loop;
      -- include pragma appends file type of TXT if a period is not at end
      -- of file name, but open fails if we put the period on there!
      if FILE_NAME(FILE_NAME_LENGTH) = '.' then
        FILE_NAME_LENGTH := FILE_NAME_LENGTH - 1;
      end if;
      FILE_IO.OPEN_INPUT_FILE(INPUT_FILES(FILE_LEVEL + 1).FILE,
                             OPENED, FILE_NAME, FILE_NAME_LENGTH);
      if not OPENED then
        TEXT_IO.PUT("ERROR IN OPENING INCLUDE FILE");
        TEXT_IO.NEW_LINE;
      else
        FILE_LEVEL := FILE_LEVEL + 1;
      end if;
    end if;
  end if;
end PUSH_FILE;
```

TEST PLAN (V. 1.0)

```
        INPUT_FILES(FILE_LEVEL).CURRENT_LINE.LENGTH := 0;
        INPUT_FILES(FILE_LEVEL).CURRENT_LINE.COLUMN := 0;
    end if;
end if;
end if;
end PUSH_FILE;
```

```
--
-- T E S T  7
--
-- Ada Style Checker Test program
with TEXT_IO; use TEXT_IO;
with SYSTEM;
with UNCHECKED_DEALLOCATION;
PRAGMA OPTIMIZE;
pragma STUPID_PRAGMA;

PROcedure ADATEST is
    type X is record
        dum1 : integer;
        DUM2 : positive;
    end record;
    ~ %
begin
    PUT("Test Program");
    NEW_LINE;
    X := CALL;
    x := "Line too little indented!";
    LONGVARIABLENAMEWITHOUTUNDERCORES := "Line too far indented.";
END ADATEST;
```


DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

CONFIGURATION MANAGEMENT PLAN

**CM Plan
03/15/88
Version 1.0**

**Written By
Leslie Vanator**

Table of Contents

1. Introduction.....	1
1.1. Scope.....	1
1.2. Acronyms.....	1
1.3. References.....	1
2. Configuration Management.....	1
2.1. Organization.....	1
2.2. Responsibilities.....	1
2.2.1. CCB.....	1
2.2.2. CM Manager.....	2
2.2.3. Document Specialist.....	2
2.2.4. Principal Architect.....	2
2.3. CMP Implementation.....	2
2.3.1. CCB.....	2
2.3.2. Release 1.0.....	2
2.3.3. Test And Support Tools.....	2
2.3.4. Policy Implementation.....	3
2.4. Applicable Policies, Directives, and Procedures.....	3
2.4.1. Release Policy.....	3
2.4.1.1. Document Releases.....	3
2.4.1.2. Code Releases.....	3
2.4.2. Access Policies.....	3
2.4.2.1. Documentation.....	3
2.4.2.2. Source Code.....	4
2.4.2.3. Object Files.....	4
2.4.2.4. Executable Image.....	4
2.4.3. Correspondence.....	4
3. Configuration Management Activities.....	4
3.1. Configuration Identification.....	4
3.1.1. Configuration Items.....	4
3.1.2. Configuration Locations.....	5
3.1.3. Current Release Repository.....	5
3.1.4. Backup and History Repository.....	5
3.1.5. Change Indication.....	5
3.1.5.1. Documentation.....	5
3.1.5.2. Code.....	5
3.2. Configuration Control.....	5
3.2.1. Change Control Documents.....	5
3.2.2. Change Request Processing.....	6
3.2.3. Discrepancy Report Processing.....	6
3.2.4. CCB Activities.....	6
3.2.5. Change To Code Repositories.....	6
3.2.6. Code Protection.....	7
3.3. Configuration Status Accounting.....	7
3.4. Audits and Reviews.....	7
4. Records Collection and Retention.....	7
A. Revision History.....	7

CONFIGURATION MANAGEMENT PLAN (V. 1.0)

Appendix 1. Structure and Contents of Directory USER\$DISK:[C474A14].....	9
Appendix 2. Structure and Contents of Directory USER\$DISK:[C474A12].....	10
Attachment 1. DASC Change Request Form.....	11
Attachment 2. DASC Discrepancy Report Form.....	12

1. Introduction

The purpose of this plan is to define all policies related to the configuration management of the Documented Ada Style Checker (DASC) Project. This plan is structured according to the DASC Documentation Standards and Guidelines.

1.1. Scope

This plan provides the methods for managing configuration items during the modification and testing of the Style Checker. These items are listed in section 3.1.1. Included are provisions for configuration identification, control, and status accounting.

1.2. Acronyms

The following acronyms are referred to in this plan:

CCB	Change Control Board
CM	Configuration Management
CMP	Configuration Management Plan
CR	Change Request
DR	Discrepancy Report
DASC	Documented Ada Style Checker
QA	Quality Assurance

1.3. References

The following documents are referred to in this plan:

- Coding Standards
- Documentation Standards and Guidelines

The above documents are part of the DASC Project and can be obtained from the DASC Project group.

2. Configuration Management

Configuration Management organization, responsibilities, plan implementation, and policies are outlined in this section.

2.1. Organization

The Configuration Management (CM) Manager will be responsible for the implementation of this plan where it applies to the Style Checker code, and to any other files necessary for execution of the Style Checker code. The Document Specialist will be responsible for the implementation of this plan where it applies to the Style Checker documentation. The CM Manager and the Document Specialist must report to the Change Control Board (CCB).

The CCB consists of the Principal Architect, the Project Administrator, the Document Specialist, and the CM Manager.

2.2. Responsibilities

The CCB, CM Manager, Document Specialist and Principal Architect have specific responsibilities to be carried out in accordance with this plan.

2.2.1. CCB

The CCB has the following responsibilities.

CONFIGURATION MANAGEMENT PLAN (V. 1.0)

- reviewing Change Requests (CR)s and Discrepancy Reports (DR)s
- approving or rejecting proposed changes
- providing for the proposal of alternative changes when necessary
- maintaining separate files of Waived, Repaired, Pending, and new Change Requests and Discrepancy Reports

2.2.2. CM Manager

The CM Manager has the following CM responsibilities

- writing the Configuration Management Plan
- creating forms necessary for CM activities
- maintaining a central repository for the current release of the configuration items 8 through 12 of section 3.1.1
- preparing releases of configuration items 8 through 12 of section 3.1.1
- notifying team members of new releases and how to access them
- tracking and recording approved changes
- maintaining a record of CM activities
- participating in CCB meetings

2.2.3. Document Specialist

The Document Specialist has the following CM responsibilities

- maintaining current releases of configuration items 1 through of section 3.1.1
- making approved changes to baselined documents
- making document copies available to team members
- maintaining each document's revision history (see Documentation Standards and Guidelines, Revision History)
- participating in CCB meetings

2.2.4. Principal Architect

The Principal Architect has the following CM responsibilities.

- calling and participating in CCB meetings
- making change control decisions
- publishing a weekly summary of all actions taken

2.3. CMP Implementation

This section outlines the CMP implementation schedule.

2.3.1. CCB

The Change Control Board has already been established and will hold meetings as Change Requests and Discrepancy reports are received.

2.3.2. Release 1.0

Release 1.0 of configuration items 1-7 of section 3.1.1 will be made by the Document Specialist after each item passes a formal review. Each baselined document will reside on floppy disk kept by the Document Specialist. The first release of the Style Checker code has been made and resides in the VAX directory [C474A14]. Every release made will reside in this location while it is the current release. Future releases will be made by the CM Manager when they are warranted.

2.3.3. Test And Support Tools

CONFIGURATION MANAGEMENT PLAN (V. 1.0)

Test and support tools are being developed independently of this plan by the team members who will be using them.

- Ada Program Library Manager (ACS): used by the CM Manager
- Manpower (COCOMO estimator), Ada Statement Counter: used by the Project Administrator
- McCabe's Complexity Measure: used by the QA Manager
- A-Test tool: used by the Project Tester
- Excelsior, Compilation Order tool: used by the Project Designer
- Ada debugger, Pretty Printer (formatter): used by the Project Implementers

2.3.4. Policy Implementation

The policies outlined in this plan will be enforced after the CMP has passed formal review. This includes all policies and procedures of section 2.4 and all activities outlined in section 3.0. The tentative date of this review is Tuesday, March 1, 1988.

2.4. Applicable Policies, Directives, and Procedures

This section outlines the policies and procedures that will be followed when making releases of DASC configuration items or when accessing these items.

2.4.1. Release Policy

Release policies for configuration items 1 through 7 of section 3.1.1 will be enforced by the Document Specialist. Release policies for items 8 through 12 of 3.1.1. will be enforced by the CM Manager.

2.4.1.1. Document Releases

All DASC Project Documents (items 1 through 7 of section 3.1.1) will be baselined by the Document Specialist as each document passes formal review. Once these documents have been baselined (release 1.0), they are under change control and cannot be changed unless an approved Change Request or Discrepancy Report is issued by the CCB. Each document will be released periodically as changes are approved and revisions are made. The Document Specialist is responsible for each release, and will provide each DASC Project member with a copy of each new release.

2.4.1.2. Code Releases

All Style Checker files including source code, object code (residing in an ACS library), and executable image will be released as a unit by the CM Manager. Once released, these files are under change control and cannot be changed unless an approved Change Request or Discrepancy Report is issued by the CCB.

Releases will be made periodically as modified code is received and checked in by the CM Manager. Frequency of releases will depend on the nature and number of the changes made to the source code, and will be determined by the CM Manager. All DASC Project team members will be informed of a new release by electronic mail.

2.4.2. Access Policies

Any configuration item under control of this plan must be accessed through the CM Manager or the Document Specialist.

2.4.2.1. Documentation

Current releases of DASC Project Documentation can be accessed through the Document Specialist. Modification of a document must be made to a copy of the document. Documents which have been modified must be sent to the Document Specialist through electronic mail. The Document Specialist will then check the modified document against approved CRs or DRs for completeness and correctness of the

CONFIGURATION MANAGEMENT PLAN (V. 1.0)

modification. The appropriate CR or DR must be signed by the Document Specialist before the CR or DR is considered closed. A new release may then be prepared.

2.4.2.2. Source Code

Source code files must be "checked out" and "checked in" following the procedures below. The CM Manager will maintain a Check Out Log that contains the following: the release number (the release being modified), the name of the source file, the name of the Ada unit affected, the corresponding Change Request or Discrepancy Report number, the individual's name, the "check out" date, and the "check in" date.

Check Out: Approval must be given by the CM Manager before copying a source code file. If the copy is being made for modification purposes, an entry will be made in the Check Out Log by the CM Manager.

Check In: Source code that has been modified must be thoroughly unit tested before being returned to the central repository. The CM Manager must be notified when testing is complete. The CM Manager will then review the CR or DR corresponding to the modification that has been made, and see that the request numbers match the numbers recorded in the source file. The CR or DR will then be signed by the CM Manager and the source file will be mailed to the CM Manager. The check in date will then be entered in the Check Out Log.

2.4.2.3. Object Files

Object files may be copied by the Project tester only with the CM Manager's approval and only for the purposes of system testing. The object files will be maintained in an ACS library.

2.4.2.4. Executable Image

The system's executable image may not be copied, but may be executed at any time.

2.4.3. Correspondence

All correspondence to and from the CM Manager and the Document Specialist should be through electronic mail. This correspondence can then be kept in a binder as a log of CM activities.

3. Configuration Management Activities

CM activities include identification and maintenance of configuration items, change control, and status accounting.

3.1. Configuration Identification

During modification and testing of the Style Checker, configuration items will be maintained by the Document Specialist and the CM Manager.

3.1.1. Configuration Items

The planned DASC Project configuration items are as follows:

1. Requirements Document
2. Configuration Management Plan
3. Quality Assurance Plan
4. Coding Standards
5. Design Document
6. Test Plans
7. User Manual
8. Source Code Files
9. Object Code Files

10. Executable Image
11. Help Library File
12. Command File

3.1.2. Configuration Locations

Configuration items 1 through 7 of section 3.1.1 will be maintained by the Document Specialist on floppy disk. Two repositories will be maintained by the CM Manager for configuration items 8 through 12 of section 3.1.1. One repository will be for the current release, the other will be a backup and histories repository.

3.1.3. Current Release Repository

The current release of the Style Checker will reside in the VAX directory [C474A14]. This subdirectory will include all source code, object code, the executable image of the current release, and all modifications to the source code. For the complete structure and contents of this subdirectory, see Appendix 1.

3.1.4. Backup and History Repository

A backup and history repository will reside in the VAX directory [C474A12]. It will consist of a backup copy of the current release's ACS library (includes source and object code) and executable image. This subdirectory will also contain the ACS library of the previous release, and the executable image of that release. For the complete structure and contents of this subdirectory, see Appendix 2. All other historical releases (source code only) will be down loaded onto floppy disk and kept by the CM Manager.

3.1.5. Change Indication

All changes to DASC configuration items must be indicated by the following methods.

3.1.5.1. Documentation

All documentation is currently identified by the titles listed in section 3.1.1 (items 1 through 7). Changes to these documents will be indicated by a change in version number, and will be noted in the document's revision history (See Documentation Standards and Guidelines).

3.1.5.2. Code

All source code is currently identified by Filename.Ada and is part of release 1.0. As code is modified, files must be renamed as follows:

```
filename_n_m.ada
  where filename = previous filename
        n = release being modified
        m = number of change being made to this file, this release
```

For example, ExampleFile_1_2.ada would indicate that Release 1.0 of ExampleFile is undergoing its second modification. Internal changes to files will be recorded in the heading documentation. This will include the date the change is made, and the Change Request or Discrepancy Report number. Changes will be flagged in the code according to methods outlined in the Coding Standards document.

3.2. Configuration Control

This section outlines the process of evaluating, and approving or disapproving changes to any DASC Project configuration item. This process begins with a release of a configuration item.

3.2.1. Change Control Documents

There are two documents that will be used to process changes.

CONFIGURATION MANAGEMENT PLAN (V. 1.0)

1. Change Request - used to request a change in the current release that is an enhancement to the system.
2. Discrepancy Report - used to request a change in the current release because it does not meet requirements.

3.2.2. Change Request Processing

The following steps will be used when processing a Change Request.

1. a Change Request Form will be prepared
2. the Change Request Form will be delivered to any member of the CCB or to the Project Manager
3. the form will be numbered for identification and tracking purposes
4. the Principal Architect will call a CCB meeting
5. the CCB will evaluate the request and either
 - approve the change and mark as Pending
 - reject the change and mark as Waived
 - suggest improvements to the change - in this case the request must be Repaired and the change must be re-evaluated
6. the change will be carried out by a DASC Project member
7. the CM Manager or Document Specialist must sign Pending forms after changes are complete

3.2.3. Discrepancy Report Processing

The following steps will be used when processing a Discrepancy Report.

1. a Discrepancy Report will be prepared
2. the Discrepancy Report will be delivered to any member of the CCB or to the Project Manager
3. the form will be numbered for identification and tracking purposes
4. the Principal Architect will call a CCB meeting
5. the CCB will evaluate the report and either
 - approve the report and the correction and mark it as Pending
 - reject the report and mark it as Waived
 - assign a DASC Project member to analyze the discrepancy and prepare a correction - in this case the report must be prepared and the correction must be re-evaluated
6. the correction will be carried out by a DASC Project member
7. the CM Manager or Document Specialist must sign pending reports after corrections are complete

3.2.4. CCB Activities

The Principal Architect will call and conduct meetings of the CCB for the purpose of evaluating all proposed changes to releases of the DASC Project configuration items. These items are listed in section 3.1.1. Meeting frequency will be dictated by submission of Discrepancy Reports or Change Requests up to a maximum of two meetings per week.

At each CCB meeting, Pending Change Requests (CR)s or Discrepancy Reports (DR)s are handled first, then each new CR and DR is considered. Each CR or DR is either waived, approved, or repaired. Waived items are archived and approved items are filed as pending to track the required change. Repaired CRs or DRs must be reconsidered by the CCB in the same manner as a new CR or DR. Each CCB meeting must have a quorum of three members. Primary responsibility for decisions rests with the Principal Architect. If the Principal Architect is absent, the responsibility rests with the Project Administrator.

3.2.5. Change To Code Repositories

Code repositories will be changed by the CM Manager when a new release is warranted. The following steps will be used to update code configurations.

1. the lowest numbered release (the release residing in [C474A12.History]) is stored on floppy

CONFIGURATION MANAGEMENT PLAN (V. 1.0)

2. the current backup ([C474A12.Backup]) is moved to the [C474A12.History] subdirectory
3. the Style Checker is re-configured with the modified source code in [C474A14.StyleChanges]
4. the modified configuration is moved to the [C474A14.Baseline] subdirectory
5. a copy of the library and executable image of the new release (now in [C474A14.Baseline]) is made in the [C474A12.Backup] subdirectory

3.2.6. Code Protection

Certain privileges to the code repositories will be granted to DASC Project team members. These privileges include Read for source code files, and Execute for the executable image. Write privilege will never be given. A complete summary of directory privileges can be found in Appendices 1 and 2.

3.3. Configuration Status Accounting

The following reports will be maintained.

Check Out Log:	This online file ([C474A14]CheckOut.Log) will be kept for tracking modifications to source code files. A new log will be started for each new release.
Change Request Forms:	All Change Request Forms will be retained by the CCB, and organized by release number.
Discrepancy Reports:	All Discrepancy Reports will be retained by the CCB, and organized by release number.
Correspondence:	Electronic mail will be retained in hard copy form by the CM Manager.
Version History:	This online file ([C474A14.Baseline]Version.Doc) will be maintained by the CM Manager, and will contain information about the current baselined code. This information includes version number, date baselined, and any concerns or problems associated with the version.

3.4. Audits and Reviews

Audits and reviews of the DASC project will be conducted by the Quality Assurance (QA) Manager.

4. Records Collection and Retention

All documentation listed in section 3.3 will be retained for configuration items 8 through 12 of section 3.1.1. When a new release is made, the Check Out Log and Version History will be hard copied and kept in a binder with the Change Requests, Discrepancy Reports, and correspondence for that release. Each release of code will have its own set of these documents.

Change Requests and Discrepancy Reports will be retained for each release of each DASC Project document (items 1 through 7 of 3.1.1).

All code releases not in code repositories will be stored on floppy disk. All document releases will also be stored on floppy. These releases along with document binders will be retained until the project is completed.

A. Revision History

Version 0.1	02/09/88	Leslie Vanator
Revision 0.2	02/25/88	Leslie Vanator

CONFIGURATION MANAGEMENT PLAN (V. 1.0)

Added Revision History and Appendices.

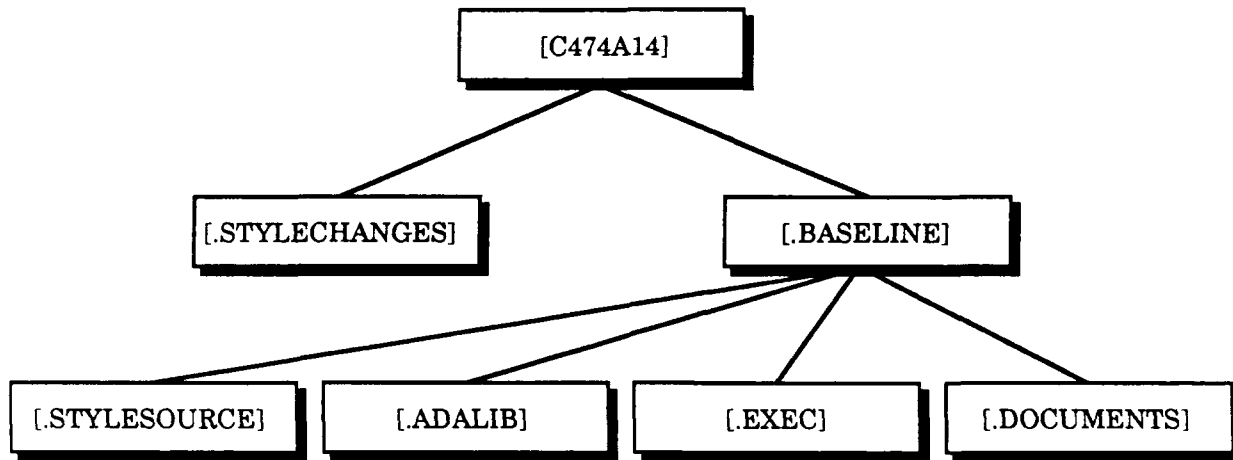
Added all material pertaining to document change control and all new information concerning the CCB.

Baselined

03/13/88

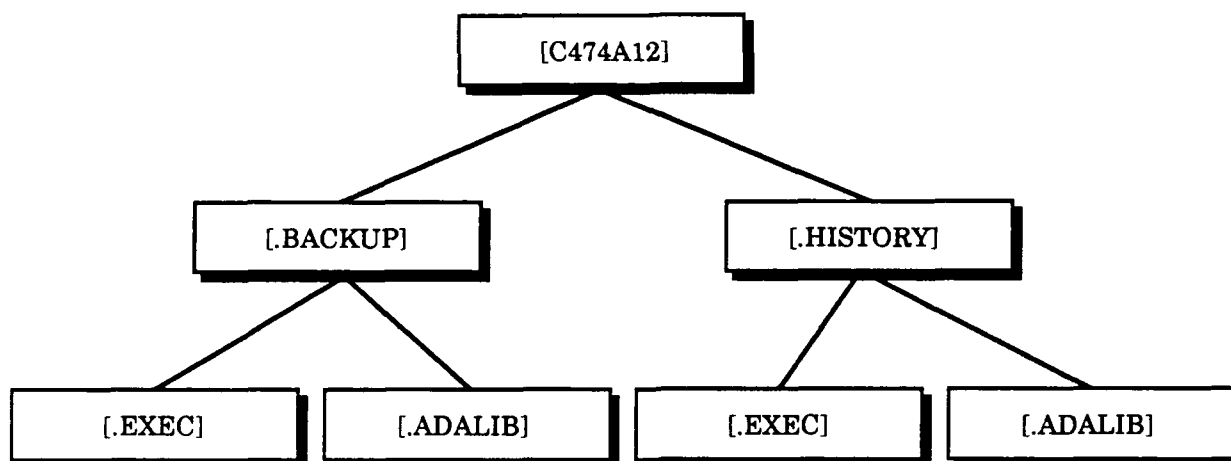
Bill Davis

Appendix 1. Structure and Contents of Directory USER\$DISK:[C474A14]



Directory Contents: USER\$DISK:[C474A14]		
SUBDIRECTORY	CONTENTS	ACCESS
C474A14	CHECK OUT LOG (CHECKOUT.LOG)	NONE
STYLECHANGES	SOURCE CODE CHANGES	READ
BASILINE	CURRENT RELEASE INFORMATION (VERSION.DOC)	READ
STYLESOURCE	CURRENT RELEASE SOURCE CODE	READ
ADALIB	CURRENT RELEASE ACS LIBRARY	NONE (READ with permission)
EXEC	CURRENT EXECUTABLE IMAGE	EXECUTE
	HELP FILE (HELP.INI)	NONE
	COMMAND FILE (STYLE.COM)	NONE
DOCUMENTS	PORTING INFORMATION (PORT.DOC)	NONE

Appendix 2. Structure and Contents of Directory USER\$DISK:[C474A12]



Directory Contents: USER\$DISK:[C474A12]		
SUBDIRECTORY	CONTENTS	ACCESS
C474A12	CM MANAGER WORK FILES	NONE
BACKUP	RELEASE INFORMATION (VERSION.DOC)	NONE
EXEC	BACKUP EXECUTABLE IMAGE	NONE
	HELP FILE (HELP.INI)	NONE
	COMMAND FILE (STYLE.COM)	NONE
ADALIB	BACKUP ACS LIBRARY	NONE
HISTORY	RELEASE INFORMATION (VERSION.DOC)	NONE
EXEC	PREVIOUS EXECUTABLE IMAGE	NONE
	HELP FILE (HELP.INI)	NONE
	COMMAND FILE (STYLE.COM)	NONE
ADALIB	PREVIOUS ACS LIBRARY	NONE

DASC CHANGE REQUEST

Change Request No.: _____
Release No.: _____

Originator: _____
EMail Address: _____

Position: _____
Date: _____

Change Type
____ New Feature ____ Cost Reduction ____ Other (describe)

Correction Description: _____

Resource Estimation (Hrs)

Modification _____
Testing _____
Other _____
TOTAL _____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ____ Waived ____ Approved With Modification ____

Reasons Waived/Description of Modification:

=====

CCB Signatures: _____

Date: _____

Request Closed

Date: _____

Configuration Manager/Document Specialist Signature: _____

DASC DISCREPANCY REPORT

Report No.: _____
Release No.: _____

Originator: _____
E-Mail Address: _____

Position: _____
Date: _____

Problem Description/Requirement Not Met: _____

Correction Description: _____

Resource Estimation (Hrs)

Modification _____
Testing _____
Other _____
TOTAL _____

Documentation Affected:

Source File(s) Affected:

CCB Decision

Approved As Is ____ Waived ____ Approved For Analysis ____

Reasons Waived:

=====

CCB Signatures:

Date:

Request Closed

Date:

Configuration Manager/Document Specialist Signature:

DOCUMENTED ADA STYLE CHECKER (DASC) PROJECT

USER MANUAL

**User Manual
04/30/88
Version 1.0**

**Written By
Bill Davis**

Table of Contents

1. Introduction.....	1
2. Purpose.....	1
3. Operation.....	1
3.1. Run Style Checker.....	2
3.2. List/View Test Files.....	4
3.3. List/View Flaw Reports.....	8
3.4. List/View Style Reports.....	11
3.5. Run Help Utility.....	14
3.6. Exit.....	17
A. Revision History.....	17

1. Introduction

This User Manual is designed to assist the user with the operation of the Documented Ada Style Checker (DASC). DASC is implemented on a Digital Equipment Corporation (DEC), Virtual Address eXtended (VAX) architecture system. This document will be useful ONLY if the user is using this DEC system. The Style Checker itself is not system-dependent. However, the user interface, which this manual treats, is completely dependent upon the DEC Command Language (DCL). Additionally, ALL references to the printing of files are unique to the facilities at Indiana University - Purdue University at Fort Wayne (IPFW).

2. Purpose

Objectively speaking, what is program style and how can it be measured? There are tools to measure program efficiency, tools to measure program complexity, and tools to measure program cost. There is a need for a tool that can measure program style. Program style has been defined as a "followed convention with respect to punctuation, capitalization, and typographic arrangement and display." DASC is a software tool that takes a syntactically correct Ada program, checks that program against an established convention, and makes a quantitative and objective evaluation of that input program. DASC is simple to use, has a built-in help facility, and will give the user output that is easily understood.

3. Operation

This Operation Section of the User Manual is subtitled according to the choices displayed by the DASC Main Menu Screen.

- (1) Run Style Checker
- (2) List/View Test Files
- (3) List/View Flaw Reports
- (4) List/View Style Reports
- (5) Run Help Utility
- (6) Exit

Each option will be described in detail. In addition, a specific input file (TEST1.ADA) with the corresponding output files (TEST1.FLW and TEST1.STY) is included.

The Style Checker is invoked at the VAX system prompt (\$) with the following entry.

```
$ @ DASC <RETURN>
```

For the sake of consistency, this document will display ALL sample user input in UPPERcase. However, DCL is not case-sensitive to any user input. Therefore, the following entries are equivalent:

```
$ @ Dasc <RETURN>
$ @ DAsc <RETURN>
$ @ daSC <RETURN>
```

The result of any of the above entries is the following DASC Introductory Screen:

DOCUMENTED ADA STYLE CHECKER (DASC) INTERFACE

This command file may be used as an interface for the Style Checker. It allows the user to (1) Run the Style Checker, (2) List and view available test files, (3) List and view flaw reports that have been generated by the Style Checker, (4) List and view style reports that have been generated by the Style Checker, (5) Run the help utility. You may enter your own file name at anytime provided the input file is syntactically correct Ada code.

USER MANUAL (V. 1.0)

To run multiple test files, create a file named "Inputfile.Ada". Inputfile.Ada must contain one or more file names with the extension "Ada". These files may be test files or your own Ada files. Each file name must be entered on a separate line. After submitting "Inputfile.Ada" to the Style Checker, the flaw report ".FLW " and style report ".STY" are created. These files will have the prefix of the first file name in "Inputfile.dat".

WARNING: There are some defaults in the interface selection. From the main menu, any key not defined with a selection number will cause the selection (1) Run Style Checker to be executed. From any yes/no inquiry, (y/n) responses other than "Y" or "y" will result in a no response.

<Press RETURN to continue>

Press the <RETURN> key to display the DASC Main Menu Screen:

DASC MENU

- 1) Run Style Checker
- 2) List/View Test Files
- 3) List/View Flaw Reports
- 4) List/View Style Reports
- 5) Run Help Utility
- 6) Exit

Select option 1 through 6

3.1. Run Style Checker

To run the Style Checker (option 1), the user must choose the following:

1 <RETURN> or
<RETURN>

The second method (as stated in the DASC Introductory Screen) chooses the default from the DASC Main Menu Screen. EITHER entry will display something similar to the following screen:

The following are test files that may be evaluated by the StyleChecker.
You may choose a test file(s) or enter your own file(s).

USER\$DISC: [XNNNXN]

TEST1.ADA;1	TEST1A.ADA;1	TEST1B.ADA;1
TEST2.ADA;1	TEST3A.ADA;1	TEST4.ADA;1
TEST5.ADA;1	TEST6.ADA;1	TEST7.ADA;1

Total of 9 files.

Enter filename with '.ADA' extension : ==> _

A user input of

TEST1.ADA <RETURN>

would result in the following screen:

Do you wish to enter another file name? (y/n) : _

NOTE: Multiple files could be entered at this time until the user had entered all the input files to be Style Checked. A user input of Y would prompt the user for another file name. However, all the flaw reports and all the style reports would be concatenated into one flaw report and one style report respectively. This method is NOT suggested if the user would like to view the reports separately.

A user input of

N <RETURN>

would result in the following screen:

The StyleChecker is now working. Please be patient.

<<while the StyleChecker is working, the following message has not yet appeared>>

The flaws report is contained in TEST1.FLW
Do you wish to view the flaw file? (y/n) : _

At this point, the output from the DASC run on input file TEST1.ADA could be viewed or printed. However, these options are essentially the same as options 3 and 4 from the main menu and will be discussed in Sections 3.3 and 3.4 respectively.

A user input of

N <RETURN>

to the prompts

Do you wish to view the flaw file? (y/n): _

and

Do you wish to view the style file? (y/n): _

would result in the following additional prompt:

Would you like to have a hardcopy of this file? (y/n) : _

A user input of

N <RETURN>

to these prompts would result in the return to the DASC Main Menu Screen.

3.2. List/View Test Files

To list or view the test input files (option 2), the user must choose the following from the DASC Main Menu Screen:

2 <RETURN>

USER MANUAL (V. 1.0)

This entry will display something similar to the following screen:

The following are test file(s) that may be used as test data for the StyleChecker.

Directory USER\$DISC: [XNNNXN]

TEST1.ADA;1	TEST1A.ADA;1	TEST1B.ADA;1
TEST2.ADA;1	TEST3A.ADA;1	TEST4.ADA;1
TEST5.ADA;1	TEST6.ADA;1	TEST7.ADA;1

Total of 9 files.

Do you wish to view a file? (y/n): _

A user input of

Y <RETURN>

would result in the additional prompt:

Enter file name with '.ADA' extension --> _

The user input of

TEST1.ADA <RETURN>

would result in the following output to the screen.

NOTE: Any screen output longer than one page will require pressing the <RETURN> key. This enhancement has been added to prevent information from scrolling past the user before he can view that output.

```
--
--  T E S T   1
--
-- This module is used to test the Coding Standards Checker for ADA - The
-- tests that are to be applied to the module are Test A1 indices 1 thru 7
-- and 14. Various additions have been added to a module originally named
-- Menu.ada authored by Jerry Baskette. Changes were made January 30, 1985
-- by James Rea.
--
-- Additions and changes have been commented throughout the module. These
-- comments relate the tests to the code.
```

```
with CURRENT_EXCEPTION;
separate (HELP_UTILITY)
```

```
-----
-- The reserved word 'procedure' is in upper case
-- This should be flagged as improper case - Test A1 index 1
-----
```

PROCEDURE PRINT_TOPIC_MENU (NODE: in HELP_LINK) is

```
    LINE_COUNT:          TEXT_RANGE := 0;  -- number of lines to be output
    TOTAL_NUMBER_OF_TOPICS: NATURAL := 0;
```

USER MANUAL (V. 1.0)

```
-- The name NUM_TOPICS_IN_COLUMN_ONE in the wrong case
-----
-- This should be flagged as improper case - Test A1 index 2
-----

num_TOPICS_IN_COLUMN_ONE: POSITIVE := 1;
TOPICS_IN_COLUMN_TWO:      BOOLEAN := FALSE;
OUTPUT_LINE:               HELP_INFO_TEXT_LINE;
OUTPUT_BUFFER:             SAVED_TEXT; -- buffer of OUTPUT_LINES to be output
RIGHT_COLUMN_START:        INTEGER := (MAX_LINE_LENGTH/2) + 2;
-----

-- The name EVEN should flag the short word (length 4)
-----

EVEN:                      BOOLEAN := FALSE;
-----

-- The name 'EVENS' should not flag the short name defn. Test A1 index 6
-----

EVENS:                     POSITIVE := 1;
-----

--The names 'I,J,K' should not be flagged. Test A1 index 7
-----

I:                          POSITIVE;
J:                          POSITIVE;
K:                          POSITIVE;
TESTFLAG:                   BOOLEAN;
TEST_FLAG:                  BOOLEAN;
-----

--The name 'TSTFLG' should be flagged as an abbreviation
--Test A1 index 14
-----

TSTFLG:                     BOOLEAN;
TEST_FLAG:                  BOOLEAN;
TESTFLAGS:                  BOOLEAN; --should be flagged for indentation
TEST_FLAGS:                 BOOLEAN; --should be flagged for indentation
CURRENT_NODE:               HELP_LINK := null;--flag for indentation
COL_ONE_NODE:               HELP_LINK := null;--flag for indentation
COL_TWO_NODE:               HELP_LINK := null;--flag for indentation

begin

-----

--Indentation should be checked per Test A2
-----

CURRENT_NODE := NODE.SUBTOPICS;
-- count the number of subtopics for this node

while CURRENT_NODE /= null loop
```

USER MANUAL (V. 1.0)

```
TOTAL_NUMBER_OF_TOPICS := TOTAL_NUMBER_OF_TOPICS + 1;
CURRENT_NODE := CURRENT_NODE.NEXT_TOPIC;
end loop;

-- If there is more than one topic, then split the topics into two columns
-- Column one will have the first half and column two the second half.
-- If there are an odd number of topics, column one will have the odd number

-----
--The following 'if' statements should be flagged as two statements
--on one line.  Test A2 index 1
-----

if TOTAL_NUMBER_OF_TOPICS /= 0 then if TOTAL_NUMBER_OF_TOPICS >= 2 then
  TOPICS_IN_COLUMN_TWO := TRUE;
  NUM_TOPICS_IN_COLUMN_ONE := TOTAL_NUMBER_OF_TOPICS / 2;

  -- More than one topic, split the number
  --See if odd number. If so, increment the topic count so odd goes in 1st col.

  if TOTAL_NUMBER_OF_TOPICS /= (TOTAL_NUMBER_OF_TOPICS/2) * 2 then
    NUM_TOPICS_IN_COLUMN_ONE := NUM_TOPICS_IN_COLUMN_ONE + 1;

  else

    -----
    --A second test of multiple statements on one line.
    -----

    EVEN := TRUE; end if; end if; --three statements on this line!

    -- set the beginning node for each column

    COL_ONE_NODE := NODE.SUBTOPICS;
    CURRENT_NODE := NODE.SUBTOPICS;

    -----
    --The following 'for' statement should flag improper indentation
    --Test A2 index 3
    -----

    for I in 1..NUM_TOPICS_IN_COLUMN_ONE loop
      COL_TWO_NODE := CURRENT_NODE.NEXT_TOPIC;
      CURRENT_NODE := CURRENT_NODE.NEXT_TOPIC;
    end loop;

    -----
    --The following 'while' and 'if' statements should be flagged for
    --improper indentation. Test A2 index 3
    --Lack of blank line following the above 'for' loop should be flagged.
    --This lack of blank line preceding the 'if' below should also
    --should also be flagged. Text A2 indices 8 & 9 respectively.
    -----

    if TOPICS_IN_COLUMN_TWO then
      while COL_TWO_NODE /= null loop
        OUTPUT_LINE := BLANK_LINE; -- blank the line buffer

        -- Put first topic in left half of output line
```

USER MANUAL (V. 1.0)

```
OUTPUT_LINE(1..COL_ONE_NODE.NAME_LENGTH) :=
  COL_ONE_NODE.NAME(1..COL_ONE_NODE.NAME_LENGTH);

-- Put second topic in second half of output line

  OUTPUT_LINE(RIGHT_COLUMN_START..RIGHT_COLUMN_START +
COL_TWO_NODE.NAME_LENGTH - 1) :=
  COL_TWO_NODE.NAME(1..COL_TWO_NODE.NAME_LENGTH);

-- Put the line in the output buffer. LINE_COUNT is incremented automatically

-----
--This is a block of comments to check the proper indentation of
--the block of comments. This is done in accordance with Test A2
--index 6. This should be flagged.
-----

  HELP_INFO_SUPPORT.APPEND_TO_DISPLAY(OUTPUT_LINE,OUTPUT_BUFFER,LINE_COUNT);
  COL_ONE_NODE := COL_ONE_NODE.NEXT_TOPIC;
  COL_TWO_NODE := COL_TWO_NODE.NEXT_TOPIC;
end loop;
end if;

if not EVEN then

-- Put the odd topic in the output buffer

  OUTPUT_LINE := BLANK_LINE;
  OUTPUT_LINE(1..COL_ONE_NODE.NAME_LENGTH) :=
    COL_ONE_NODE.NAME(1..COL_ONE_NODE.NAME_LENGTH);
  HELP_INFO_SUPPORT.APPEND_TO_DISPLAY(OUTPUT_LINE,OUTPUT_BUFFER,LINE_COUNT);
end if;

-- Now print the output buffer

  HELP_INFO_SUPPORT.PRINT_TEXT(OUTPUT_BUFFER,LINE_COUNT);
end if;

-----
--Lack of blank lines separating the major block (exception) should
--be flagged. Test A2 index 7.
-----

exception
  when others => text_io.put_line("Print Menu " & CURRENT_EXCEPTION.NAME);
  raise;
end PRINT_TOPIC_MENU;
```

After the file has been displayed on the screen, the following prompt will appear :

Would you like to have a hardcopy of any file? (y/n): _

A user input of

N <RETURN>

would return the user to the DASC Main Menu Screen.

USER MANUAL (V. 1.0)

A user input of

Y <RETURN>

would result in the additional prompt:

Enter file name with '.ADA' extension ==> _

A user input of

TEST1.ADA <RETURN>

would result in the following DASC Printer Destination Screen:

```
IPFW Printer job controller
```

Bldg.	ID	Printer Desc.	Location	Access	Pickup
Kettler	(1)	LPAO bulk printer	Computer Room	open	box
	(2)	LPBO upper/lowercase	Computer Room	Pascal users	box
	(3)	LETTER word processing	Room 204A	50 block max.	self-serv
	(5)	LCAO graphics/text	Computer Room	Graphics users	box
	(8)	LASER Laser Printer	8 A.M.- 5 P.M.	hi quality copy	C&DP Off.
	(9)	LIAO Up/LO case bulk	Computer Room	open	box
	(10)	KTOPEN upper/lowercase	Kettler cluster	open	self-serv
Neff	(6)	NEFF graphics/text	B74	Graphics users	self-serv
Library	(7)	LIBRARY small-size jobs	1st Floor	50 block max.	self-serv
Union	(4)	UNION small-size jobs	Room 125	50 block max.	self-serv

Enter the number of the desired destination and a message will appear stating that your job has been placed into the print queue at that destination.

You will now be returned to the DASC Main Menu Screen.

3.3. List/View Flaw Reports

To list or view the flaw report files (option 3), the user must choose the following from the DASC Main Menu Screen:

3 <RETURN>

This entry will display the following screen (if ALL test input files have been run by the StyleChecker):

The following file(s) are flaw reports that the StyleChecker has generated.

Directory USER\$DISC: [XNNXN]

TEST1.FLW;1	TEST1A.FLW;1	TEST1B.FLW;1
TEST2.FLW;1	TEST3A.FLW;1	TEST4.FLW;1
TEST5.FLW;1	TEST6.FLW;1	TEST7.FLW;1

Total of 9 files.

Do you wish to view a file? (y/n): _

A user input of

Y <RETURN>

would result in the additional prompt:

Enter file name with '.FLW' extension ==> _

The user input of

TEST1.FLW <RETURN>

would result in the following output to the screen.

NOTE: Any screen output longer than one page will require pressing the <RETURN> key. This enhancement has been added to prevent information from scrolling past the user before he can view that output.

```
with CURRENT_EXCEPTION;
This package is on the list of packages to be warned against.
```

```
PROCEDURE PRINT_TOPIC_MENU (NODE: in HELP_LINK) is
Reserve word PROCEDURE should be in lower case
```

```
num_TOPICS_IN_COLUMN_ONE: POSITIVE := 1;
Object name num_TOPICS_IN_COLUMN_ONE should be in upper case
```

```
TESTFLAGS:                                BOOLEAN; --should be flagged for indentation
This line should be indented to column: 5
```

```
TEST_FLAGS:                                BOOLEAN; --should be flagged for indentation
This line should be indented to column: 5
```

```
if TOTAL_NUMBER_OF_TOPICS /= 0 then if TOTAL_NUMBER_OF_TOPICS >= 2 then
There are more than one statements on this line!
```

```
EVEN := TRUE; end if; end if; --three statements on this line!
There are more than one statements on this line!
```

```
if TOTAL_NUMBER_OF_TOPICS /= 0 then if TOTAL_NUMBER_OF_TOPICS >= 2 then
This structure should have following blank lines to set it off.
```

```
COL_ONE_NODE := NODE.SUBTOPICS;
This line should be indented to column: 2
```

```
CURRENT_NODE := NODE.SUBTOPICS;
This line should be indented to column: 2
```

```
for I in 1..NUM_TOPICS_IN_COLUMN_ONE loop
This line should be indented to column: 2
```

```
if TOPICS_IN_COLUMN_TWO then
This line should be indented to column: 2
```

```
OUTPUT_LINE := BLANK_LINE; -- blank the line buffer
Beginning of this block not indented properly. Line ignored for Indentation!
```

USER MANUAL (V. 1.0)

OUTPUT_LINE(1..COL_ONE_NODE.NAME_LENGTH) :=
Beginning of this block not indented properly. Line ignored for Indentation!

COL_TWO_NODE.NAME_LENGTH - 1) :=
The statement-continuation in this line should be indented!

while COL_TWO_NODE /= null loop
This structure should have preceding blank lines to set it off.

while COL_TWO_NODE /= null loop
This structure should have following blank lines to set it off.

while COL_TWO_NODE /= null loop
This structure is large enough that it should have a loop-name!

if TOPICS_IN_COLUMN_TWO then
This structure should have following blank lines to set it off.

if not EVEN then
This line should be indented to column: 2

if TOTAL_NUMBER_OF_TOPICS /= 0 then if TOTAL_NUMBER_OF_TOPICS >= 2 then
This structure should have following blank lines to set it off.

when others => text_io.put_line("Print Menu " & CURRENT_EXCEPTION.NAME);
Object name text_io should be in upper case

when others => text_io.put_line("Print Menu " & CURRENT_EXCEPTION.NAME);
Object name put_line should be in upper case

PROCEDURE PRINT_TOPIC_MENU (NODE: in HELP_LINK) is
This structure should have following blank lines to set it off.

--
Finish checking started in above statement. Total statements: 66

After the file has been displayed on the screen, the following prompt will appear:

Would you like to have a hardcopy of any file? (y/n): _

A user input of

N <RETURN>

would return the user to the DASC Main Menu Screen.

A user input of

Y <RETURN>

would result in the additional prompt:

Enter file name with '.FLW' extension ==> _

A user input of

TEST1.FLW <RETURN>

would result in the DASC Printer Destination Screen (shown on page 8).

Enter the number of the desired destination and a message will appear stating that your job has been placed into the print queue at that destination.

You will now be returned to the DASC Main Menu Screen.

3.4. List/View Style Reports

To list or view the style report files (option 4), the user must choose the following from the DASC Main Menu Screen:

4 <RETURN>

This entry will display the following screen (if ALL test input files have been run by the StyleChecker):

The following file(s) are style reports that the StyleChecker has generated.

Directory USER\$DISC: [XNNNXN]

TEST1.STY;1	TEST1A.STY;1	TEST1B.STY;1
TEST2.STY;1	TEST3A.STY;1	TEST4.STY;1
TEST5.STY;1	TEST6.STY;1	TEST7.STY;1

Total of 9 files.

Do you wish to view a file? (y/n): _

A user input of

Y <RETURN>

would result in the additional prompt:

Enter file name with '.STY' extension ==> _

The user input of

TEST1.STY <RETURN>

would result in the following output to the screen.

NOTE: Any screen output longer than one page will require pressing the <RETURN> key. This enhancement has been added to prevent information from scrolling past the user before he can view that output.

STYLE Report
USER\$DISK: [XNNNXN] TEST1.ADA;1

Naming Conventions

*	Invalid Case for an Object Identifier	3	Errors
*	Invalid Case for a Keyword	1	Error
	Name Segment Size (Separated	Desired	<5
	by Underscores)	Actual	4.8
	Average Name Size	Desired	>5
		Actual	10.2
			Characters

USER MANUAL (V. 1.0)

Physical Layout

*	Occurrences of More Than One Statement/Line	2	Errors
*	Inconsistent Indentation	14	Errors
*	Missing Blank Lines to Set Off a Block	3	Errors
*	Loops Without Names	1	

Information Hiding, Abstraction, Data Use

*	Percent of Literals In Body	Desired	< 30.0	
		Actual	71.4%	
	Percent of Universal Types	Desired	< 40.0%	
		Actual	9.5%	
!	Data Structuring Types NOT Used	Array Types		
		Enumeration Types		
		Record Types		
!	No Attributes are Used			
!	Ada-Specific Features NOT used	AND THEN		
		OR ELSE		
		EXITS		
		XOR		
		ELSIF		
		OUT parameters		
		IN OUT parameters		
		PRIVATEs		

Modularity

Average Number of Parameters	Range	0..4	Parameters
Instances of parameters below minimum		0	
Instances of parameters above maximum		0	
Average Subprogram Size	Range	10..200	Statements
Instances of size below minimum		0	
Instances of size above maximum		0	
Loops with too many exit statements		0	Instances
Control Structure Nesting	Maximum	8	
	Exceeded	0	Instances
Package Nesting	Maximum	2	
	Exceeded	0	Instances
Subprogram Nesting	Maximum	4	
	Exceeded	0	Instances

Comment Usage

	Number of Comments	83	Comments
	Average Comment Size	Desired >15	Characters
*		Actual 53.0	Characters

Transportability

Number of Lines Exceeding Line Length	0
Address Clauses	0
Representation Specifications	0
PRAGMA'S used:	

USER MANUAL (V. 1.0)

Non-Standard PRAGMA's Used
Packages/Procedures WITHed

CURRENT_EXCEPTION

* => Style Flaw ! => Note: Potential for improvement

Keyword Usage				
Used Keyword	Allowed	Restriction	Occurrences	Percentage
<hr/>				
IF	yes	0.0%	10	19.6%
IN	yes	0.0%	2	3.9%
IS	yes	0.0%	1	2.0%
END	yes	0.0%	9	17.6%
FOR	yes	0.0%	1	2.0%
NOT	yes	0.0%	1	2.0%
ELSE	yes	0.0%	1	2.0%
LOOP	yes	0.0%	6	11.8%
NULL	yes	0.0%	5	9.8%
THEN	yes	0.0%	5	9.8%
WHEN	yes	0.0%	1	2.0%
WITH	yes	0.0%	1	2.0%
BEGIN	yes	0.0%	1	2.0%
RAISE	yes	0.0%	1	2.0%
WHILE	yes	0.0%	2	3.9%
OTHERS	yes	0.0%	1	2.0%
SEPARATE	yes	0.0%	1	2.0%
EXCEPTION	yes	0.0%	1	2.0%
PROCEDURE	yes	0.0%	1	2.0%

After the file has been displayed on the screen, the following prompt will appear:

Would you like to have a hardcopy of any file? (y/n): _

A user input of

N <RETURN>

would return the user to the DASC Main Menu Screen.

A user input of

Y <RETURN>

would result in the additional prompt:

Enter file name with '.STY' extension ==> _

A user input of

TEST1.STY <RETURN>

would result in the DASC Printer Destination Screen (shown on page 8).

Enter the number of the desired destination and a message will appear stating that your job has been placed into the print queue at that destination.

You will now be returned to the DASC Main Menu Screen.

3.5. Run Help Utility

The DASC on-line help facility contains 53 different topics. However, there are four levels of nesting and all topics are NOT accessible from EVERY level. An alphabetical listing of each nesting level along with the minimum character string needed to uniquely select that topic appears at the end of this section. For example, if an "I" or "IN" was entered from the top level, BOTH INSTALLATION AND INDIVIDUAL_PARAMETERS topics would be selected. But an entry of "INS" would select ONLY topic INSTALLATION.

NOTE: The help facility utilizes the <RETURN> key to upward exit from a particular nesting level. Therefore, the user should locate the <NO SCROLL> key and utilize this key to prevent topics with multiple pages from scrolling past the viewing screen too quickly.

To Run the help utility (option 5), the user must choose the following from the DASC Main Menu Screen:

5 <RETURN>

This entry will display the following screen:

The help utility/Style Checker is now working. Please be patient. Locate the <no scroll> key and press to prevent information from scrolling past.

<< there is a slight delay at this point while the help utility is being loaded >>

HELP

This is the Help Facility for the Style Checker. More information on specific topics may be obtained by entering the leading portion of the name of any of the topics.

To get the list of topics available at any time, enter a ? at the prompt.

To list the information on all topics below the current topic, enter a * at the prompt.

To exit the Help Facility, enter a <CR> for each level of information.

Additional Information Available:

INSTALLATION STYLE_ISSUES_IN_GENERAL

OPERATION MAINTENANCE

INDIVIDUAL_PARAMETERS

HELP subtopic ? _

A user input of

IND <RETURN>

would result in the following screen:

INDIVIDUAL_PARAMETERS

The parameters defining the limits of the Ada style are defined in the body of the FILE_HANDLING package. In that body the default values are specified as defaults for the individual parameter variables. There is also a procedure SET_STYLE_PARAMETERS which sets the actual values for the style.

It is expected that any local changes to the style parameters will be done in the SET_STYLE_PARAMETERS procedure, so that the original defaults remain unchanged.

Additional Information Available:

ERRORS_TO_LIST	PERCENT_UNIVERSAL
OUTPUT_KEYWORD_LIST	DATA_STRUCTURES
SHORT_PROGRAM	ATTRIBUTE_CHECK
...	...
...	...
COMMENT_SIZE	KEYWORD_PARAMETERS
PERCENT_LITERALS	

HELP INDIVIDUAL_PARAMETERS subtopic ? _

A user input of

E <RETURN>

would result in the following screen:

ERRORS_TO_LIST

To avoid redundancy repeating instances of detected errors, the 'errors-to-list' parameter restricts the number of times any one error is listed. This means, for example, only the first 5 times the user forgot to use loop-names would be listed. Other occurrences of each individual error would be counted, and the total instances of the error would be noted on the style summary.

HELP INDIVIDUAL_PARAMETERS subtopic ? _

A user input of

<RETURN>

would "upward exit" one level of the help facility and result in the following prompt:

HELP subtopic ? _

A user input of

<RETURN>

would "upward exit" one level of the help facility and result in the following prompt:

topic ? _

A user input of

<RETURN>

would return the user to the DASC Main Menu Screen.

The following is a listing by nesting level of ALL available help topics in DASC. The top levels are left justified. The letter(s) in parentheses are the minimum character(s) that can be entered to uniquely select the topic on that level.

```

INDIVIDUAL_PARAMETERS (IND)
  ADDRESS_CLAUSE (AD)
  ATTRIBUTE_CHECK (AT)
  AVE_NAME_LEN (AV)
  BLANK_LINES (B)
  CHARACTER_SET (CH)
  COMMENT_SIZE (COM)
  CONTROL_NESTING (CON)
  DATA_STRUCTURES (D)
  ERRORS_TO_LIST (E)
  INDENT_COMMENTS (INDENT_C)
  INDENT_TYPES (INDENT_T)
  KEYWORD_PARAMETERS (K)
  LINE_SIZE (LI)
  LOOP_NAMES (LO)
  NUMBER_OF_LOOP_EXITS (N)
  OBJECT_CASE (OB)
  OUTPUT_KEYWORD_LIST (OU)
  PACKAGE_NESTING (PA)
  PERCENT_LITERALS (PERCENT_L)
  PERCENT_UNIVERSAL (PERCENT_U)
  PRAGMAS (PRA)
  PREDEFINED_PRAGMA (PRE)
  PROSCRIBED_PACKAGE (PRO)
  REPRESENTATION_SPECS (REP)
  RESERVED_CASE (RES)
  SHORT_PROGRAM (SHORT_P)
  SHORT_STRUCTURE (SHORT_S)
  SHORT_WORD (SHORT_W)
  SPELLING_REQUIRED (SP)
  SUBPROGRAM_NESTING (SUBPROGRAM_N)
  SUBPROGRAM_PARAMETERS (SUBPROGRAM_P)
  SUBPROGRAM_SIZE (SUBPROGRAM_S)
  UNDERSCORES (UNDERSCORES)
  UNDERSCORE_SIZE (UNDERSCORE_)
  VOWEL_FREQUENCY (V)
INSTALLATION (INS)
  COMMAND_FILE (COMM)
  COMPILATION (COMP)
  SPECIFIC_FILE_NAMES (S)

MAINTENANCE (M)
OPERATION (O)
  INPUT_FILES (I)
  OUTPUT_FILES (O)
    FLAWS_FILE (F)
    REPORT_FILE (R)

```

USER MANUAL (V. 1.0)

COMMENT_USAGE (C)
INFORMATION_HIDING (I)
MODULARITY (M)
NAMING_CONVENTIONS (N)
PHYSICAL_LAYOUT (P)
TRANSPORTABILITY (T)

STYLE_ISSUES_IN_GENERAL

3.6. Exit

To exit from the DASC Main Menu Screen (option 6) and return to the system prompt (\$), the user must choose the following.

6 <RETURN>

However, from ANY point in the DASC operation, the user can cause an interrupt in the executing program. The DEC system will immediately return any user to the system prompt (\$) with the following procedure:

<CTRL> Y <<< press simultaneously >>>

This emergency exit procedure concludes the DASC User Manual. It is hoped that the Style Checker has been a useful software tool for evaluating program style.

A. Revision History

Version 0.1	04/24/8	Bill Davis
Revision 1.0	04/30/88	Bill Davis
	Corrected spelling and punctuation errors.	
	Standardized representation for <RETURN>.	

Diskette Order Form

Machine-readable source code (including the test suite) and documentation for the DASC software system are available from the SEI. To receive a set of the distribution diskettes, please select the desired format below, and return this form with \$10.00 payment to:

Software Engineering Curriculum Project
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Checks may be made payable to **Carnegie Mellon University** and should accompany this order form.

Source code format:

- ☐ Macintosh diskette (3.5", double sided, 800K byte)
- ☐ PC/AT-compatible diskette (5.25", double sided, high density, 1.2M byte)

Documentation format:

- ☐ Macintosh diskette (includes *Microsoft Word*, *MacWrite*, and text-only formats)
- ☐ PC/AT-compatible diskette (text-only format)

Send to:

Name _____

Address _____

UNLIMITED, UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS NONE													
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED													
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A		5. MONITORING ORGANIZATION REPORT NUMBER(S)													
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-EM-1		7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE													
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.		7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE HANSCOM, MA 01731													
6b. OFFICE SYMBOL (If applicable) SEI		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003													
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE		10. SOURCE OF FUNDING NOS. <table border="1"><tr><td>PROGRAM ELEMENT NO.</td><td>PROJECT NO.</td><td>TASK NO.</td><td>WORK UNIT NO.</td></tr><tr><td>63752F</td><td>N/A</td><td>N/A</td><td>N/A</td></tr></table>		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.	63752F	N/A	N/A	N/A				
PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.												
63752F	N/A	N/A	N/A												
8b. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		11. TITLE (Include Security Classification) Software Maintenance Exercises for a Software Engineering Project Course													
12. PERSONAL AUTHOR(S) Charles B. Engle, Gary Ford, Tim Korson															
13a. TYPE OF REPORT FINAL		13b. TIME COVERED FROM _____ TO _____													
14. DATE OF REPORT (Yr., Mo., Day) February, 1989		15. PAGE COUNT													
16. SUPPLEMENTARY NOTATION															
17. COSATI CODES <table border="1"><tr><td>FIELD</td><td>GROUP</td><td>SUB. GR.</td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></table>		FIELD	GROUP	SUB. GR.										18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) software maintenance education	
FIELD	GROUP	SUB. GR.													
19. ABSTRACT (Continue on reverse if necessary and identify by block number) <p>Software maintenance is an important task in the software industry and thus an important part of the education of a software engineer. It has been neglected in education, partly because of the difficulty of preparing a software system upon which maintenance can be performed. This report provides an operational software system of 10,000 lines of Ada and several exercises based on that system. Concepts such as configuration management, regression testing, code reviews, and stepwise abstraction can be taught with these exercises.</p>															
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION													
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF		22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630													
		22c. OFFICE SYMBOL SEI JPO													

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Software Engineering Curriculum Project is developing a wide range of materials to support software engineering education. A *curriculum module* (CM) identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in designing a course. A *support materials package* (SM) contains materials related to a module that may be helpful in teaching a course. An *educational materials package* (EM) contains other materials not necessarily related to a curriculum module. Other publications include software engineering curriculum recommendations and course designs.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

Permission to make copies or derivative works of SEI curriculum modules, support materials, and educational materials is granted, without fee, provided that the copies and derivative works are not made or distributed for direct commercial advantage, and that all copies and derivative works cite the original document by name, author's name, and document number and give notice that the copying is by permission of Carnegie Mellon University.

Comments on SEI educational materials and requests for additional information should be addressed to the Software Engineering Curriculum Project, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Electronic mail can be sent to education@sei.cmu.edu on the Internet.

Curriculum Modules (* Support Materials available)

- CM-1 [superseded by CM-19]
- CM-2 Introduction to Software Design
- CM-3 The Software Technical Review Process*
- CM-4 Software Configuration Management*
- CM-5 Information Protection
- CM-6 Software Safety
- CM-7 Assurance of Software Quality
- CM-8 Formal Specification of Software*
- CM-9 Unit Testing and Analysis
- CM-10 Models of Software Evolution: Life Cycle and Process
- CM-11 Software Specifications: A Framework
- CM-12 Software Metrics
- CM-13 Introduction to Software Verification and Validation
- CM-14 Intellectual Property Protection for Software
- CM-15 Software Development and Licensing Contracts
- CM-16 Software Development Using VDM
- CM-17 User Interface Development*
- CM-18 [superseded by CM-23]
- CM-19 Software Requirements
- CM-20 Formal Verification of Programs
- CM-21 Software Project Management
- CM-22 Software Design Methods for Real-Time Systems*
- CM-23 Technical Writing for Software Engineers
- CM-24 Concepts of Concurrent Programming
- CM-25 Language and System Support for Concurrent Programming*
- CM-26 Understanding Program Dependencies

Educational Materials

- EM-1 Software Maintenance Exercises for a Software Engineering Project Course
- EM-2 APSE Interactive Monitor: An Artifact for Software Engineering Education
- EM-3 Reading Computer Programs: Instructor's Guide and Exercises